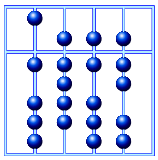


Lehrstuhl für Netzwerkarchitekturen
Fakultät für Informatik
Technische Universität München

Viable Network Intrusion Detection in High-Performance Environments

Robin Sommer

Vollständiger Abdruck der von der Fakultät für Informatik
der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Alfons Kemper, Ph.D.
Prüfer der Dissertation:	1.	Univ.-Prof. Anja Feldmann, Ph.D.
	2.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
	3.	Adj. Prof. Vern Paxson, Ph.D. University of California, Berkeley / USA (schriftliche Beurteilung)

Die Dissertation wurde am 30. 06. 2005 bei der Technischen Universität München
eingereicht und durch die Fakultät für Informatik
am 29. 08. 2005 angenommen.

Abstract

Network intrusion detection systems (NIDS) continuously monitor network traffic for malicious activity, raising alerts when they detect attacks. However, high-performance Gbps networks pose major challenges for these systems. Despite vendor promises, they often fail to work reliably in such environments.

In this work, we set out to understand the trade-offs involved in network intrusion detection, and we mitigate the impact of their choice on operational security monitoring. We base our study on extensive experience with several large-scale network environments, including the Munich Scientific Network and the backbone of the University of California at Berkeley. In such networks, we find an immense traffic diversity which requires a NIDS to deal robustly with unexpected situations. However, to accommodate any conceivable situation, a NIDS would need an unlimited supply of CPU cycles and memory. Thus, the operator of the system needs to trade-off the quality of the detection with resource demands. To provide the necessary tuning options, we devise several new mechanisms which allow to choose this trade-off according to the policy of a particular environment. Moreover, we enable a NIDS to transparently share its state across instances, thereby multiplying the available amount of resources. Another major trade-off that a NIDS faces is the decision when to alert: if it reports anything which could potentially be malicious, it will generate an unmanageable number of alerts; if it reports only the most obvious attacks, it will miss some. To improve the precision of the detection, we enable a NIDS to incorporate different kinds of network context into its analysis. Such contextual information can either be derived during operation or provided externally, e.g., by host applications.

The thesis starts with recapitulating concepts and limitations of network intrusion detection, and then presents seven operational environments in which we have conducted our research. Next, we discuss our experiences with deploying four different NIDSs, open-source systems as well as commercial. We examine the resource usage of one of the systems in detail and improve the system's resource management. Then we introduce the concept of *independent state*, i.e., internal fine-grained state that can be shared among instances. The implementation of this concept provides us with a wealth of new operationally significant applications. Finally, we introduce *contextual signatures* which supplement traditional byte-level pattern matching with contextual information. We implement and evaluate all of our improvements within the framework provided by the open-source Bro NIDS.

Acknowledgments

I would like to thank the many people who have contributed to this thesis. Without them, this work would not have been possible.

First and foremost, I am indebted to my advisor, Anja Feldmann, for the outstanding motivation, guidance, support, and knowledge she has provided throughout the course of this work. She kindly took me into her group and provided me with a great amount of freedom to work on things that I like. She has become a good friend, and her feedback has greatly improved the thesis.

I am extremely grateful to Vern Paxson for the assistance, generosity, and advice that I have received from him. I have benefited greatly from his insight, support, and feedback. His enthusiasm has always been a great motivation, and I enjoyed my stays at ICSI very much.

My gratitude also extends to Manfred Broy and Alfons Kemper for agreeing to be on my thesis committee.

I also thank my colleagues and friends in our group at TU München. It has been fun working with them. In particular, many thanks go to Holger Dreger. His ideas, efforts, and contributions have significantly added to my work. Thanks to Vinay Aggarwal and Jörg Wallerich for commenting on drafts of the thesis, to Jörg also for mastering FreeBSD, to Petra Lorenz for helping with bureaucracy, and to Gregor Maier for his help with the DAG cards.

I also thank Scott Campbell for regular feedback and test-driving of my code; José María González for analyzing BPF-based sampling; Christian Kreibich for his terrific work on Broccoli and his related contributions; Wolfgang Müller for arranging the opportunity to start the work; and to the people at ICSI for their help during my visits.

My work has been generously supported by the following institutions providing me with access to their network environments: the International Computer Science Institute, the Lawrence Berkeley National Laboratory, the Leibniz-Rechenzentrum, the National Energy Research Scientific Computing Center, the Saarland University, and the University of California at Berkeley. Without the help of many great people working at these places, this work would have been impossible. In particular, I thank Victor Apostolescu of the Leibniz-Rechenzentrum for his outright support. Moreover, my thanks go to Mike Bennett, Christopher Chin, Mark Dedlow, Deti Fliegl, Jake F. Harwood, Mike Hunter, David Johnson, Jay Krous, Alfred Läpple, Ken Lindahl, Dietmar Reichert, Helmut Tröbs, Claus Wimmer, as well as to LBNL's BroLite group.

Last, but not the least, I thank my parents, my sister, as well as my grandparents. It has been their lasting love and support that enabled me to reach this point.

Contents

List of Figures

1 Introduction

> *What are the false positive rates [of product Foo] like*? Very low. I've seen situations where other products [...] generated as much as 800,000 false positives a day in one particular environment, and with [Foo], around 100,000. Still a lot but a big improvement.
>
> — *Posting to an intrusion detection mailing list in 2005 [ML].*

1.1 Motivation

Network intrusion detection systems (NIDS) are a major security component in many network environments. These systems continuously monitor network traffic for malicious activity, raising alerts when they detect attacks. However, while vendors promise effective and efficient network protection, in operational deployments the merits of NIDSs are regularly questioned: on the one hand, they miss actual attacks; on the other hand, they tend to overwhelm the analyst with irrelevant alerts. Moreover, despite vendor promises, in larger networks NIDSs often lack the processing performance required to analyze the traffic stream to a sufficient depth.

Obviously, there is a gap between claims and operational reality. The roots of such practical problems have hardly seen any investigation yet. Commercial vendors sell black-boxes without providing the customer with any details of internal mechanisms—which makes it very hard to comprehend the actual abilities and short-comings of such a system. Researchers, on the other hand, do publish results, yet rarely have access to large-scale operational networks to verify practicability.

In this thesis, we set out to bridge the gap by analyzing potentials and limitations of network intrusion detection in large real-world networks. We base our work on extensive experience with operational networks of different scales, including highly-saturated Gbps backbones. We analyze, evaluate, and improve current network intrusion detection technology, particularly focusing on its viability in high-performance environments. It turns out that the applicability of theoretical concepts is often restricted by practical restrictions; approaches which are perfectly reasonable in small networks do not scale to environments with tens of thousands of users and hosts.

Our work is motivated by the experience of setting up the open-source Bro NIDS in a medium-scale university environment. When we set it up for the first time, problems arose immediately. The system usually exhausted the host's memory after running a couple of hours, and it missed a significant fraction of the network data as its processing was not able to keep up with the packet stream in real-time. When we experimented

with the most popular open-source system, Snort, we were faced with similar problems: while Snort did not run out of memory, it did drop a large fraction of the packets as well. Moreover, Snort produced an immense number of alerts, many of which quickly turned out be wrong. Later, we saw that commercial systems show similar effects as Bro and Snort do.

Upon closer examination, it turns out that all NIDSs face fundamental trade-offs between detection rate on the one hand and resource demand as well as false positive rate on the other. Choosing these trade-offs is an environment-specific decision. When we examined different networks—multiple large university and research-lab environments—we noticed that the demands of these networks are very diverse. On the one hand, their security policies have different focuses. On the other hand, their traffic patterns differ significantly. Hence, the trade-offs they require vary as well. However, NIDSs are often lacking mechanisms to tune their behavior to a chosen trade-off. Therefore, we developed several new mechanisms to customize a NIDS accordingly.

In particular, large high-performance environments (Gbps networks with tens of terabytes of external traffic a month), pose major challenges for a NIDS. Due to the volume and heterogeneity of their traffic, short-term characteristics are essentially unpredictable. Thus, a NIDS needs to deal robustly with situations which are vastly different from any kind of "average".

Moreover, existing NIDSs are not exploiting their full potential to mitigate the effect of the trade-offs they face. One problem is that most NIDSs do not really cooperate. Even if a NIDS supports the installation of multiple detectors inside a network, these components typically work independently, only exchanging high-level results such as alerts. At the same time, each detector maintains a great deal of internal state which remains hidden from the others. Yet, combining all available knowledge promises to improve the detection rate without impacting either resource demands or false positive rate.

Another problem is that almost all of a NIDS's false positives are due to the system lacking "background information", i.e., the system observes symptoms which can be due to either benign or malicious activity, and it cannot reliably decide which is the case. Including more context in the matching process helps to increase the detection quality significantly. The problem becomes most apparent when examining the predominant approach to network intrusion detection: "signature matching" uses byte-level patterns of known attacks. The network traffic is scanned for these patterns, and an alert is raised if any is found. However, raw packet-bytes are often not sufficient to assess activity reliably—that may require more information about, e.g., the end-hosts participating in the conversation.

While our observations hold for network intrusion detection in general, our primary object of study is the Bro NIDS, a powerful and flexible open-source research system running on commodity hardware. Throughout our work, we analyze and enhance this system, using it as a vehicle for new functionality that we develop. In the context of this system, we considerably improve the state management, make resource demands more dynamic, share state among multiple instances, interface the system to external applications, and develop a pattern matching engine which leverages network context to avoid false alerts. All of our extensions are already included in the official Bro distribution.

1.2 Outline

We now briefly summarize the following chapters.

Chapter 2. In the second chapter, we start with recapitulating basic concepts of network intrusion detection, introducing our terminology as we go. One focus of the discussion is the theoretical and practical limitations of network intrusion detection. Moreover, we present a selection of available NIDSs, among them four systems that we have deployed: the open-source systems *Snort* and *Bro* as well as the commercial systems *Enterasys Dragon* and *McAfee IntruShield*.

Chapter 3. In the third chapter, we introduce seven network environments in which we have conducted our research; among them the backbones of the *Munich Scientific Network* and the *University of California at Berkeley*. We discuss main characteristics of large networks, focusing on policies, threats, and traffic characteristics.

Chapter 4. In the fourth chapter, we report our experiences with deploying NIDSs. We discuss their logging/alerting facilities, signature quality, user interfaces, resource demands, and evaluation difficulties. We observe traffic artifacts introduced by the monitoring infrastructure as well as the particular sensitivity which NIDSs exhibit with regards to programming errors. Then we focus on the resource requirements of one of the NIDSs, the Bro system. We analyze its processing and memory demands in-depth, finding that some of its design assumptions are impractical in high-performance networks. Based on this observation, we devise several enhancements to the Bro system which eventually make it suitable for operational use in all of the discussed environments.

Chapter 5. In the fifth chapter, we greatly extend NIDS functionality by introducing *independent state*, i.e., internal fine-grained state that can be propagated between NIDS instances. We implement a unified architecture for independent state within the Bro system which is able to transparently make all of the system's internal state accessible to other instances running concurrently or subsequently. We present several applications such as distributed processing, load parallelization, persistent state, crash recovery, debugging support, and integration of host-based context. A performance evaluation shows that our architecture is suitable for use in high-performance environments.

Chapter 6. In the sixth chapter, we turn to improving detection quality. Most NIDSs detect intrusions by means of signatures, i.e., byte-level patterns of known attacks. While highly efficient, such signatures lack the tightness required to avoid false alarms. We introduce *contextual signatures* which supplement traditional-style signature matching with additional context on different levels of abstraction. Therewith, we greatly extend the expressiveness of signatures and hence the ability to reduce false alerts. We implement contextual signatures within the Bro system, and verify the performance of our implementation by comparing the new matching engine to the most-widely used open-source system, Snort.

Chapter 7. In the last chapter, we summarize the thesis, recurring our main observations and contributions. We close with a discussion of promising future work.

1.3 Published Work

Parts of this thesis have been published:

Robin Sommer and Vern Paxson
Exploiting Independent State For Network Intrusion Detection
Proc. 21st Annual Computer Security Applications Conference, 2005

Holger Dreger, Christian Kreibich, Vern Paxson, and Robin Sommer
Enhancing the Accuracy of Network-based Intrusion Detect. with Host-based Context
Proc. Confer. on Detection of Intrusions and Malware & Vulnerability Assessm., 2005

Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer
Operational Experiences with High-Volume Network Intrusion Detection
Proc. 11th ACM Conference on Computer and Communications Security, 2004

Robin Sommer and Vern Paxson
Enhancing Byte-Level Network Intrusion Detection Signatures with Context
Proc. 10th ACM Conference on Computer and Communications Security, 2003

Robin Sommer
Bro: An Open Source Network Intrusion Detection System
Proc. 17. DFN-Arbeitstagung über Kommunikationsnetze, 2003

2 Network Intrusion Detection

Facts do not cease to exist because they are ignored.

— *Aldous Leonard Huxley*

In this chapter, we recapitulate basics of network intrusion detection. After summarizing the historic origins of intrusion detection in §2.1, we introduce general security terminology in §2.2. Next, §2.3 presents basic concepts of network intrusion detection, followed by a discussion of theoretical and practical limitations in §2.4. The chapter finishes with a description of some of the available systems in §2.5.

2.1 Historic Origins of Network Intrusion Detection

Intrusion detection[1] originates from traditional *audit* systems: in early computing environments, large mainframes produced chronological records of system events which could then be examined manually for purposes such as accounting and security. In the 1970s, the U.S. Department of Defense (DOD) outlined security goals for such audit mechanisms, among which were allowing the discovery of attempts to bypass protection mechanisms. Later, James P. Anderson first articulated the goal of automatic audit *reduction* to remove irrelevant records. From 1984 to 1986, Dorothy Denning and Peter Neumann developed a theoretical model of a real-time intrusion detection system, the *Intrusion Detection Expert System (IDES)* [Den87] which uses a combination of statistical metrics and models. IDES was implemented from 1986-1992 at *SRI International* and served as the basis for many future research efforts. The *Network System Monitor (NSM)* [SBD+91], developed in 1991 at the University of California at Davis, was the first intrusion detection system which used network packets as its primary data source, rather than OS-level records. Hence, it was the first *network* intrusion detection system. In 1992, a collaboration of the U.S. Air Force, UC Davis and others developed the *Distributed Intrusion Detection System (DIDS)* which for the first time physically distributed detection and analysis across multiple locations, integrating host-based and network-based analysis.

[1] This section is based on Bace [Bac00] who draws a detailed time-line of security audit and intrusion detection.

2.2 Terminology

We now introduce general terminology used throughout this thesis. We note that related literature sometimes defines these terms with slightly varying semantics.

2.2.1 Computer Security

Security is not an asset in itself but a property of a system. The basic components of computer security are *confidentiality*, *integrity*, and *availability* [Bis03]. Confidentiality ensures that only authorized persons may access information. Integrity guarantees that information is trustworthy. Availability assures that people are able to access information they are authorized for.

2.2.2 Policy and Mechanism

For a particular system, the definition of security depends on the deploying site's characteristics. A certain use of a system may be illegal at one site yet legitimate at another. Therefore, before discussing security, one needs to define a *policy*: "a *security policy* is a statement of what is, and what is not, allowed" [Bis03].

We refer to the security policy of a particular site as the *site policy*. In principle, such a site policy could be formally defined using a mathematical calculus. However, such models tend to get complex and unmanageable in practice. Thus, more often policies are given informally as plain text statements (e.g., often sites have terms-of-use which users have to sign before being granted access; [Bis03] shows an example). The major drawback of these informal policies is their lack of completeness: many details are subject to interpretation.

Components which are deployed to enforce security need to know what to look for. Thus, devices are configured with *device policies*.[2] Ideally, device policies match with the site policy—but in reality they usually do not, primarily because device policies *need* to be formalized (e.g., by means of a configuration language). As long as the site policy is only specified informally, inevitably there are mismatches between the two.

A device policy is different from the *mechanism* that a device employs: "a mechanism is a method, tool, or procedure for enforcing a policy" [Bis03]. While this distinction is crucial, in practice many security devices intermingle the two. Often, device mechanisms establish device policies which cannot be customized. We refer to policies which are implied by a mechanism as *implicit device policies*. If, on the other hand, a device policy is customizable by the device's administrator, we call it an *explicit device policy*. When devising a security mechanism, introducing implicit policies should be avoided as it is often hard to comprehend *what* they in fact imply. Yet, in practice device policies usually include both implicit and explicit aspects.

[2] Bace calls them *monitoring policies* [Bac00].

2.2.3 Threats, Vulnerabilities, Intrusions

If a site policy has been defined, a *threat* is a "potential violation" [Bis03]. For the violation to actually occur, there need to be one or more *vulnerabilities*. These are "weaknesses in systems that can be exploited in ways that violate security policy" [Bac00]. An *intrusion* (or synonymously *attack*) is an attempt to exploit a vulnerability. An intrusion may be *successful* or *unsuccessful*. The person who conducts the intrusion is the *intruder* (or *attacker*).

From a security perspective, one strives to *minimize* the threats, *fix* the vulnerabilities, and *prevent* the intrusions. If preventing an intrusion has not been possible, at the very least one needs to *detect* it. If an intrusion has been successful, one needs to *recover*. A recovery requires a *forensic analysis*, i.e., analyzing all compromised systems to understand the causes and the consequences of an intrusion.

In order to organize and analyze intrusions systematically, several taxonomies of intrusions have been suggested in the literature. One of the most well-known is presented in [NP89]. The authors categorize intrusions into the categories *external misuse*, *hardware misuse*, *masquerading*, *setting up subsequent misuse*, *bypassing intended control*, *active misuse of resources*, *passive misuse of resources*, *misuse resulting from inaction*, and *use as an indirect aid in committing other misuse*. [LJ97] refines the taxonomy by further subdividing categories and adding a second dimension for characterizing an intrusion's result: *exposure*, *denial of service*, and *erroneous output*. Still, such schemes are often too general to effectively categorize a concrete intrusion. The authors of [NP89] themselves observe that their categories are not mutually exclusive. This problem is amplified by the fact that intrusions are often composed of multiple steps, each of which may fit into a different category. Furthermore, while examining the outcome of an experiment, the authors of [LJ97] found that all of the detected intrusions fitted into just three of the nine categories (which was their motivation to further subdivide the three groups).

Given these problems, we use the more pragmatic classification scheme suggested in [HH04]. The authors, using a CERT's point of view, focus mainly on network-based attacks and classify intrusion along the dimensions *attack vector*, *attack target*, *exploited vulnerability*, and *effect beyond the attack itself*. The attack vector dimension is further subdivided; the sub-categories are shown in Table 2.1. In general, a given incident may apply to more than one of these sub-categories (or to none at all). In such a case, the authors suggest to take the "best match"—which is deliberately fuzzy. For the vulnerability dimension, the identifiers of the *Common Vulnerabilities and Exposures Initiative* [CVE] are leveraged. CVE numbers are the de-facto standard for naming software vulnerabilities. They are maintained by the MITRE corporation [MIT], and are supported by a large number of free and commercial security systems. Alternative ways to reference publicly know vulnerabilities are the numbers of US-CERT's *Technical Cyber Security Alerts* (formerly *CERT advisories*) and *Bugtraq IDs* [Bug].

We introduce two more dimensions of classifying attacks. First, we differentiate between *directed* and *undirected* attacks. Directed attacks are intentionally aimed at a selected victim. Undirected attacks, on the other hand, are executed against arbitrarily

Table 2.1 Attack vector categories (adapted from [HH04]).

Attack Vector Category	Description
Virus	Self-replicating program that needs manual actions to distribute.[a]
Worm	Self-replicating program that distributes without any manual action.
Trojan	Program made to appear benign that serves some malicious purpose.
Buffer Overflow	Process that gains control or crashes another process by overflowing the other process' buffer.
Denial of Service	Attack which prevents legitimate users from accessing or using a host or network.
Network Attack	Attack focused on attacking a network or the users on the network by manipulating network protocols.
Physical Attack	Attack based on damaging physical components of a network or computer.
Password Attack	Attack aimed at gaining a password.
Information Gathering Attack	Attack in which important information is gained by the attacker but no physical or digital damage is done. Most importantly, a *scan* probes the victim for certain properties such as offered services.

[a] Compared to [HH04], we removed the dependency on files in the distinction between *worm* and *virus*. Instead, we differentiate the two by the need for manual actions as suggested in [WPSC03].

chosen systems which may happen to be vulnerable. Second, we call attacks which are scripted to run unattended, *automated attacks*. In contrast, *manual attacks* require the attacker to perform some of the actions interactively. Automated attacks are faster and easier to perform than manual ones. In particular, once an attack has been automated, it can be conducted by people who lack the knowledge required to perform the manual attack. Informally, such persons are commonly referred to as *script kiddies*. In the Internet, most of the attacks are currently automated and undirected; we return to this point in §3.3.2.

2.3 Concepts of Network Intrusion Detection

In this section, we briefly discuss the main concepts underlying network intrusion detection, citing relevant related work as we proceed. Due to the large body of intrusion detection research, our discussion of related literature is inevitably incomplete, and we refer the reader to the bibliographies of the given sources for further links.

Due to the widespread prevalence of the Internet protocol suite [Ste94], all major current network intrusion detection systems are built for IP networks. Consequently, we focus on such environments as well.

2.3.1 Definitions

An *intrusion detection system* attempts to detect intrusions. In this thesis, we focus on network-based systems, i.e., *network intrusion detection systems (NIDS)*, which we define as systems whose *primary source of data is network traffic*. In contrast, there are *host intrusion detection systems (HIDS)* which rely on information gathered on individual hosts. *Hybrid* systems are both network- and host-based.

When a NIDS believes that it has detected an intrusion, it raises an *alert*. A human analyst can then inspect the alert to assess its severeness. If the system has correctly identified an intrusion, the alert is a *true positive*. If the NIDS alarms although no intrusion has occurred, it is a *false positive*. On the other hand, if the NIDS fails to report an actual intrusion, we encounter a *false negative*. If it correctly remains quite when no intrusion has taken place, we see a *true negative*. We note that these definitions depend on the site policy rather than the NIDS's device policy. In particular, an alert is a false positive if it reports a violation of the NIDS's device policy which is not part of the site policy.

Conceptually, NIDSs often do not make their decisions based on individual network packets but on *events* which are (policy-neutral) abstractions of network activity. Typical events include the establishment of a connection or the download of a file. If an event (or a sequence of events) violates a site policy, the NIDS should trigger an alert.

2.3.2 Architecture

In general, a NIDS consists of different components. In the literature, the discussion of components often already implies details of the detection method which need not hold in general.[3] We abstract from such specifics and have identified five types of components (see Figure 2.1):

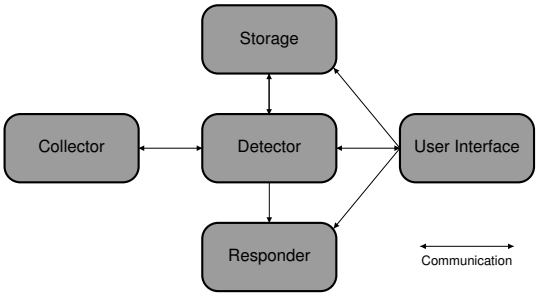
Collector. Provides an interface for accessing data that is used by the detection process. For a NIDS, the primary kind of data collector is a *network tap*. A tap provides access to all raw network packets which cross a particular position of a network. Other types of collectors include interfaces to host-based data or external databases.

Detector. Conducts the actual detection process. The detector is the "brain" of the NIDS. It accesses data provided by collector and storage (see below), and it decides what should trigger an alert.

User Interface. Reports results to the user, and enables the user to control the NIDS. In particular, the user interface is needed to define the NIDS's device policy. In the simplest case, the user interface consists of ASCII files. More sophisticated systems often include graphical user interfaces (GUIs).

[3] For instance, [Bac00] includes a *pattern matcher* in the "Generic Intrusion Detection System"

Figure 2.1 General architecture of a network intrusion detection system.



Storage. Stores persistent data required by the detector or the user interface.[4] Such data is either derived by the detector itself or provided externally. Often, a storage leverages a database system.

Responder. Reacts to detected intrusions in order to prevent future damage. Active responses may include dropping the connectivity to the potential attacker or even counter-attacks. A response may be triggered automatically or manually via the user interface.

All of these components may exist more than once. On the other hand, it is quite rare that any is missing except for the responder. The components can be combined into a single piece of software as well as be logically or physically separated. If separated, the components include communication sub-systems to exchange information.

In principle, a detector may analyze input either in *batches* or in *real-time*. Batched processing was common in early NIDSs which lacked the resources required for continuous monitoring. Nowadays, all major systems work in real-time, and hence in this thesis we only consider real-time detection. We note, however, that batched analysis is still common for host-based analysis; e.g., log files are often scanned for malicious activity at regular intervals.

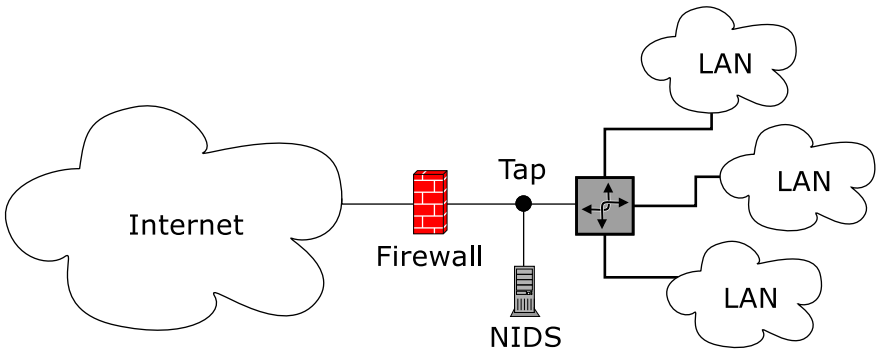
2.3.3 Deployment Schemes

With the network tap being the primary data collector, a NIDS is usually deployed at one or more central network points. The specific locations are chosen according to the types of intrusions that are to be detected. Figure 2.2 shows a very basic but common scheme which is potentially able to detect attacks from the external Internet. A collector is installed at the network's exclusive upstream link. Thus, the detector is able to examine all incoming and outgoing traffic. However, it cannot see any packets exchanged between internal hosts.[5]

If there are multiple upstream links, multiple collectors are required. In such installations each collector is often combined with a detector which performs a preliminary

[4] In the case of an successful intrusion, data provided by the storage may be valuable for forensic analysis

Figure 2.2 Typical deployment scheme of a network intrusion detection system.



analysis. It reduces the volume of the data and forwards its condensed results to a central detector. Such a combined collector/detector is a *sensor*. Accordingly, we refer to the schemes that use multiple sensors and one central detector as the *sensor model*. The main advantage of the sensor model is that, while multiple network locations are monitored, there is still a primary detector which has a comprehensive view of the network. In particular, by having the user interface to the central detector, a NIDS is able to hide some of the underlying complexity.

Unfortunately, due to the star-topology the sensor model does not scale well. Therefore, more widely distributed systems have been developed. *Hierarchical* installations feature a set of detectors ordered in layers. Detectors on layer n receive results from layer $n-1$, and forward their results to layer $n+1$. Often, such hierarchies are organized in trees with the collectors at the leafs, and the user interface at the root. Finally, the *freely distributed* systems do not imply any specific component layout.

2.3.4 Detection Methods

In the literature, there are plenty of different approaches for detecting intrusions. Generally, they fall into the categories *misuse detection*, *anomaly detection*, and *specification-based detection*. We subsequently briefly summarize these approaches.

Misuse Detection

Misuse detection systems identify *known* attacks. They are equipped with a library of *attack patterns*; each time such a pattern is found in the network stream, an attack is assumed to be in progress. Such a library requires expert knowledge to build, and it needs to be updated regularly to accommodate new attacks. Its content can be based

as well.

[5] An interesting question is whether to install the NIDS on the internal or the external side of the firewall. Due the large of number of scans originating in the Internet, most people prefer the variant shown in Figure 2.2.

on experiences with attacks from the past, on dissected exploits, or on anticipated but yet unseen attacks.

There are different semantic levels at which attack patterns can be described. The most common representation are specific byte sequences which are known to appear in the raw payload stream of attack connections. All major commercial NIDSs use this technique. We call these byte-level patterns *signatures*. [FV01] and [CSM01] present approaches to efficiently match hundreds of fixed strings in parallel. Higher-level attack representations include declarative languages to describe relations between events (e.g., [LP99, CO00, PD01]); state transition graphs which represents attacks as a sequence of state transitions of the monitored system (e.g., [KS94, VEK00]); and graph-based network activity descriptions [SCC+96].

The main advantage of misuse detection is precision: if patterns are tight, attacks can be reliably distinguished from benign traffic. However, in §6 we see that, in practice, finding tight patterns can be difficult. The main disadvantage of misuse detection is its inability to detect yet unknown attacks.[6]

We note that initial deployment of a misuse detection system often implies the installation of a default device policy: using a predefined attack library defines all matching traffic to be malicious, even if it is in accordance with the site policy (e.g., signature libraries often contain patterns to detect the use of certain applications). The process of explicitly—and often incrementally—adapting the pattern library to the site policy is commonly referred to as *tuning*.

Anomaly Detection

Anomaly detection systems find deviations from *expected behavior*. To this end, they are equipped with a profile of *normal* activity. If observed activity deviates significantly from the profile, an alert is raised.

In the literature, there is a wide variety of different models supposed to capture normal behavior. Many of them are *learning*, i.e., they adapt their profile over the course of time. The classic anomaly-based system is IDES [Den87] (and its successor NIDES [JV93]) which uses a combination of statistical metrics and models such as thresholds, confidence intervals, and Markov models. Since then, many more approaches have been suggested. Often, they borrow ideas from the AI community, including information theory [LX01], neural networks [JCdCM98, ZLM+01], genetic algorithms [SPM99], and artificial immune-systems [Hof99].

The underlying assumption of anomaly detection is that legitimate behavior of users (and systems) is fully predictable. Even more, each anomaly-based NIDS assumes that behavior is predictable *within the scope of its model*. Therefore, the deployment of such a system always implies installing a rigorous (and often implicit) device policy which may be difficult to match with the site policy. Even when providing explicitly tunable parameters, it may be hard to understand the impact of a concrete parameterization;

[6] There are approaches to misuse detection which use self-learning techniques to create new patterns automatically; e.g., [NKS05] uses Bayesian classifiers as well as clustering. Such schemes effectively combine misuse and anomaly detection.

e.g., in [TM] the authors discuss at length why a certain parameter of a particular host-based immune-system-inspired system needs to be the constant 6.

Consequently, the dominant problem of anomaly-based NIDSs are false positives. In practice, very simple schemes, such as thresholds for certain kinds of activity, turn out to be the most useful representatives of this approach. None of the more sophisticated models seem to be suitable for real-world deployment. However, recent research suggests that anomaly-based detection may work well for small well-defined application domains [KV03, KMVV03].

In terms of performance, anomaly-based systems are particular hard to evaluate (see §2.4.5) since *normality* is a very environment-specific term. In terms of evading the detection, an attacker may be able to leverage knowledge of the system's model to find a blind spot; he may be able to hide his attack in activity which appears to be benign [WS02, TM02].[7] Moreover, learning systems are prone to deliberate manipulation of their profiles: by executing a carefully chosen series of benign steps, an attacker may be able to gradually teach the system to accept an attack.

Specification-based Detection

Specification-based systems feature an explicit specification of *allowed behavior*. If observed activity is not covered by the specification, it is flagged as malicious.

Specification-based detection has been introduced in the context of host-based systems [KFL94, KRL97]. Later, this approach has also been applied to network-based detection [RSS99, SGF+02], although often not in its pure form ([RSS99] includes misuse-based elements; [SGF+02] leverages machine-learning techniques).

Specification-based detection is the inverse of misuse detection: it characterizes the benign activity rather than the malicious. Therefore, in principle, both are equally powerful. However, in practice, the specification-based approach tends to be less feasible, primarily because it requires defining *all* benign situations in terms of a device policy. This is equivalent to formalizing the site policy and thus rarely practicable as discussed in §2.2.2. Misuse-detection, on the other hand, can resort to covering only the *subset* of malicious activity which can be expressed by a device policy.

Nevertheless, for small well-defined application domains, specification-based detection is actually rather promising. A simple example are communication patterns (which hosts are allowed to talk to which destinations?). In networks, such patterns are often already formalized (and supposed to be enforced) by means of firewall rule-sets [CBR03].

2.3.5 Alert Correlation

In a distributed NIDS setup, a single attack may be observed and reported by multiple sensors. In this case, the administrator should, ideally, not see a couple of distinct messages but only a single comprehensive alert. Thus, a NIDS needs to perform *alert*

[7] [WS02, TM02] discuss attacks on host-based anomaly-detection systems. Yet, the ideas are applicable to network-based systems as well.

correlation (i.e., identifying related alerts) and *alert fusion* (i.e., merging related alerts).[8] Giving the large number of alerts that NIDSs tend to generate (see §4.2.1), such a post-processing is crucial.

Often, correlation needs to incorporate data from *heterogeneous* sources, such as different host-/network-based NIDSs, firewalls, and logging facilities. Due to this heterogeneity, correlation schemes first need to represent the inputs in a common *meta-format* which covers the individual device formats. There are several proprietary formats in use by commercial vendors, yet there is also an IETF working group defining a standardized *Intrusion Detection Exchange Format (IDMEF)* [IDM].

Two basic groups of correlation schemes have been proposed for network intrusion detection: schemes which require patterns of actual attacks and/or alert interdependencies, and schemes which do not. Members of the first group include [NCR02, CM02, MD03, MMDD02]. In [NCR02], alerts are correlated based on knowledge about their prerequisites and consequences defined by means of correlation graphs. In [CM02], alert correlations are identified based on attack descriptions in a special-purpose language. In [MD03], an already existing language from another domain is leveraged to describe alert relationships. A formal calculus for alert correlation is presented in [MMDD02].

The second group of correlation schemes works without any specific knowledge of actual attacks. [VS01] correlates IDMEF alerts according to probabilistic feature similarities. In [QL03], such *hyper-alerts* are correlated in time by means of a statistical time series analysis. Examining a proprietary system, [DW01] identifies duplicates, consequences (i.e., alerts that must occur jointly) as well as a set of general situations suitable for merging. For the Emerald system (see §2.5.3), [PFV02] calculates incident ranks based on priorities; then a clustering algorithm combines related incidents.

Unfortunately, despite its importance, the research on alert correlation is still at a rather early stage. The approaches in the first group need considerable predefined knowledge, facing a similar problem as misuse-detection (see §2.3.4): it is hard to detect unknown dependencies. The approaches in the second group often appear to be rather ad-hoc and heuristic; many of them simply merge alerts based on similar attributes such as affected hosts.

The main problem of alert correlation is the heterogeneity of the input. Often, the exact semantics of an individual alert are unknown, and hence a meaningful correlation is hard to achieve. This becomes apparent when looking at the meta-formats. One example is the attribute "confidence" that IDMEF alerts may include. The "confidence" gives an estimate on how certain the issuing device is about the facts that it reports. The attribute's valid range of values is the set `low`, `medium`, `high`, and `numeric` with the latter being a probability between 0 and 1. It appears to be almost impossible to correlate such ratings across different kinds of devices. In fact, it may already be hard for the *same* kind of device.

[8] In the literature, alert correlation and alert fusion are often used synonymously, and in the following we do so as well.

2.3.6 Detection vs. Prevention

Historically, the main purpose of NIDSs was to *detect* attacks. Therefore, they were called network intrusion *detection* systems, even though some NIDSs could actively react to identified attacks. For instance, they were able to deny an attacker further access by means of dynamically reconfiguring a firewall. Nowadays, most commercial vendors are selling network intrusion *prevention* systems (NIPS).

Originally, the term NIPS referred to a NIDS that is installed in-line with the network, instead of using a passive network tap. The main advantage of such an in-line system is that it can *block* traffic that it believes to be malicious. In this way, attack traffic can be prevented from reaching its destination. A non-in-line system is, at best, restricted to stop *future* packets.

Unfortunately, the term NIPS is rather blurred by now. This is mainly due to a marketing hype produced by an (in-)famous Gartner report [Sti03]. The report declared intrusion detection to be "dead" and to be replaced by intrusion prevention. As a consequence, commercial vendors hurried to re-brand their systems to prevention systems. Some did not care whether they were indeed working in-line, while others deliberately used terms slightly different from "prevention" for their passive systems (e.g., the non-inline Dragon system sells as "Enterasys Dragon Intrusion *Defense*" [Dra]).

Therefore, we avoid the term "prevention system". Instead, we call a system which follows the original intent of a NIPS an *in-line NIDS*. Thereby, we also emphasize that the most important component is still the detector. It cannot block an attack without first recognizing it. Additionally, we call a NIDS *active* if, after detecting an attack, it is able to react in a way that directly impairs the attacker. All other systems are *passive*. By definition, all in-line systems are active.

2.3.7 Stateful vs. Stateless

In principle, a NIDS can be either *stateful* or *stateless*. A stateless system considers every packet on its own, not recalling anything that it has derived in the past. On the other hand, a stateful system maintains an in-memory representation of its knowledge about the current state of the network which is regularly updated as input is processed. Such knowledge enables more reliable decisions while avoiding a large set of evasion attacks (see §2.4.3).

All major NIDSs are stateful. However, they differ in the type and amount of state they store. A common type of state is *connection state*: for every active connection, the system stores information like duration, status of the handshake, and payload volume. Other types of state include *per-host state* (such as connection attempts per source address to detect scanners) and *signature state* (for signatures that so far have only partially matched). In §4.4 we examine the state kept by one NIDS in detail.

More generally, such network state is one type of *contextual information* which a NIDS can incorporate into its analysis to support its decisions. In general, the more context a NIDS has at hand the more reliable it can assess network activity. For example, a NIDS may be provided with information about the operating systems that internal

hosts are using. Then it can adapt its analysis accordingly to avoid evasion attacks which leverage internal differences between OSs [SP03]. We return to incorporating contextual information in §5.7 (where we include host-supplied context into the analysis) and §6 (where we support signature matching with contextual information).

2.3.8 User Interface

In most environments, a deployed NIDS needs regular attention to examine the alerts and to tune the system for maximum performance. Therefore, the user interface is an important factor of the system's effectiveness.

Types of user interfaces differ as do the demands on it by the users. In the simplest—yet not necessarily least effective—case, the interface consists of ASCII files for both configuration and logging. This is often the case for open-source systems, although sometimes add-on packages are available which provide graphical interfaces (e.g., *ACID* [ACI] for Snort, §2.5.1, and *Brooery* [KS05] for Bro, §2.5.2). In contrast, all major commercial systems ship with graphical user interfaces of different degrees of sophistication. To address as large a customer-base as possible, they strive to find a balance between ease-of-use and depth of technical details. To this end, GUIs often provide different views, ranging from "executive summaries"—e.g., colorful bar-plots showing number of attacks per day—to inspection of raw packet payload.

Despite their importance, user interfaces have not yet seen much discussion in the intrusion detection research literature. In [HS01, KO04] two different approaches of visualizing Snort log files are discussed: SnortSnarf [HS01] provides an interactive hyper-linked view of aggregated log-messages; SnortView's [KO04] main tool is a two-dimensional time diagram. The authors of both works note that dealing with false alerts is the most dominant challenge. [KS05] focuses on analyzing log archive content, providing flexible filters and context-aware mechanisms for quick drill-down.

2.3.9 License Scheme

There are free open-source NIDSs (such as Snort, §2.5.1, and Bro, §2.5.2), as well as commercial closed-source systems (such as IntruShield, §2.5.6, and Dragon, §2.5.5). Arguments in favor of one license scheme over the other are well-known, and we do not repeat them here.

Nevertheless, with respect to network intrusion detection it is worth mentioning that a typical signature-based system consists of two parts for which different licenses may apply: *(i)* the implementation of the NIDS, and *(ii)* its signature library. For commercial systems, the source code of the implementation is usually not available. However, some vendors make the clear-text signatures available to their customers, enabling them to comprehend the precise semantics of alerts. Other vendors choose to keep their signatures close to avoid evasion attacks as well as reimplementations for other systems. For the open-source Snort, the system's implementation is covered by the *GNU General Public License* [GPL]. However, a large fraction of its standard signatures are subject to a more strict license which essentially forbids commercial redistribution (see §4.2.2). Yet, all of

Snort signatures are open in the sense that a user can examine them to understand their matching semantics.

From a research point of view, a closed-source product is considerably harder to evaluate since one is restricted to black-box testing. This is true both for the NIDS itself as well as for its signature library. On the other hand, commercial NIDSs are often supported by a strong company security group which has the resources to provide high-quality signatures in a timely-fashion. For community-developed open-source systems this can be difficult to achieve (see §4.2.2).

2.3.10 Hardware

When designing a NIDS, there are two basic choices for the hardware platform: commodity hardware and dedicated custom hardware. In terms of the cost/performance ratio, off-the-shelf PCs are hard to surpass. Moreover, a NIDS running on a widely-available platform can be distributed as a software-only package. Specialized hardware, on the other hand, is more powerful, yet also more expensive, and it needs to be bundled with the software.

In terms of computing power, the main requirement for a NIDS is its ability to keep up with a packet-stream in real-time. For low-volume streams, commodity hardware is sufficient. However, as the volume increases, such hardware approaches its limits. Over the years, enhancements to the OS-level infrastructures have pushed this limit further, e.g., by means of more efficient packet filtering [MJ93] and data transfers [Der03]. Yet, in Gbps environments one already has to accept occasional packet drops (see §4.4), and given that network capacities are going to increase further, at some point commodity hardware will cease to work reliably.

Most commonly, custom hardware is used as a *supplement* to conventional components. In general, the most natural candidates for hardware-support are tasks which are performed *per packet*. These include packet-capturing (and -forwarding in case of an in-line system) and signature matching. Typically, networks processors, FPGAs, or highly-specialized network interface cards (see §4.5.4) are used for such jobs. Commercial systems sometimes come in different flavors, depending on the targeted environment: while for small networks commodity hardware is sufficient, more expensive boxes targeting larger networks are extended with custom hardware.

2.4 Limitations of Network Intrusion Detection

Network intrusion detection systems face a number of fundamental limitations, both theoretical and practical, which we summarize below.

2.4.1 Cost vs. Benefit

It is expensive to effectively deploy a NIDS. Apart from the one-time cost of purchasing the system, there are two kinds of ongoing expenses. First, the software needs to be kept current. For a commercial NIDS, this usually means subscribing to an update

service. Second, and more costly, human analysts need to be employed who are capable of handling the system's output. Too often, organizations buy a NIDS without provisioning for the resources required to take care of it.

Almost always, only a certain amount of money is available to ensure a network's secure operation; and a NIDS represents only *one* component of a security concept. Where the available funding is best spent needs to be decided on a case-by-case basis. However, a cost-benefit analysis for a NIDS is difficult to achieve, and requires a thorough risk analysis. It is important to remember that, when purchasing a NIDS, one does not buy security; rather one *increases the probability* of detecting (and potentially preventing) attacks.

However, sometimes reliable detection is *not* the primary objective of a NIDS installation. For an organization, there may be regulations or term contracts which require (or suggest) to install security devices. For instance, the deployment of a NIDS could reduce insurance fees, independent of any actual improvement of security.

2.4.2 False Positives

One of the main factors limiting the use of a NIDS is the system's *false positive rate*, i.e., the ratio between false positives and true positives. Often this ratio is very high: many NIDSs tend to overwhelm the analyst with alerts which are in fact irrelevant. If an analyst notices that most of the alerts are not related to any intrusion, he will quickly "learn" to ignore *all* of them. The problem is amplified by the fact that often NIDSs are deployed without provisioning for the time an analyst needs to sift through their output—each day. The busier an analyst is, the more he will be tempted to skip the daily NIDS alerts.

The cause of false positives are mismatches between the site policy and the NIDS's (implicit or explicit) device policy. Site policies are mostly defined informally and, therefore, require the analyst to leverage a great deal of common sense. Naturally, this cannot be expected from a device. Yet, even if (a part of) the site policy is formalized, many NIDSs lack the flexibility required to parameterize their mechanisms accordingly.

In a classic article [Axe00], Axelsson showed how crucial the false positive rate is. Suppose for a moment that we can precisely measure the false positive and negative rates of a NIDS (in practice, we cannot; see §2.4.5). Suppose further that we know the number of data units[9] p that the NIDS analyzes (very large) as well as the number of units i affected by actual intrusions (very low). By applying Bayes' theorem we can then derive the probability p that a given alert does in fact indicate an intrusion. Axelsson calls p the *Bayesian detection rate*. It turns out that even with an optimal (yet unrealistic) false negative rate of 0, we need to have a very low false positive rate to get an acceptable Bayesian detection rate. For instance, with $p = 1 * 10^6$ and $i = 20$, the false positive rate needs to be on the order of $1 * 10^{-5}$ at most to get a Bayesian detection rate of 66%. Using the same parameters but a more realistic false negative rate of 0.3, p goes down to 58%, i.e., nearly half of the alerts are bogus.

[9] Such a unit may, e.g., be a packet or an event.

This issue is an instance of the classic "needle in a haystack" problem: the vast volume of benign input effectively hides the intrusions. Naturally, one can argue about the parameters used in the example above; for real environments, none of them is known. But in any case, the false positive rate is the dominant factor in terms of the Bayesian detection rate while the impact of the true positive rate is rather insignificant.

2.4.3 Attack Vulnerability

A NIDS is supposed to protect a network from attacks. Therefore, for an attacker the NIDS itself is a very natural target: if he is able to defeat the NIDS, his actual attack may go unnoticed. There are two main types of attacks against a NIDS: *evasion* attacks misleading the NIDS's processing, and *denial-of-service* attacks constricting the system in its intended processing.

Evasion Attacks

Evasion attacks are one of the most fundamental problems in network intrusion detection. They mislead the NIDS in its perception of the network stream. If a NIDS, for some reason, interprets the semantics of a connection differently than the involved endpoints, it cannot reliably detect contained attacks.

In a seminal paper [PN98], Ptacek and Newsham describe several evasion attacks[10], conducted at the transport- and networks layers of the TCP/IP protocol stack. A very simple example is sending packets with carefully chosen time-to-live values (TTLs). A NIDS captures network packets somewhere on the way between originator and responder. If a packet's TTL is chosen such that it reaches the NIDS but not the endpoint, the NIDS analyzes data which is not part of the actual payload stream. Hence, the attacker is able to sneak in additional data into this NIDS's view which may, e.g., fool its pattern matcher. Another example is ambiguities in the network stream. Consider, e.g., two overlapping IP fragments: fragment A covers the bytes from offset a to b while fragment B covers c to d with $c < b$. If the two fragments contain inconsistent data, the situation is ambiguous and not covered[11] by the IP standard [Pos81]. However, end-systems have to decide how to treat such packets, and different systems do it differently: [SP03] observes five different strategies that are in use today. If the NIDS uses another strategy than the receiving endpoint, it will be desynchronized and potentially miss attacks. At the application-layer, there are similar problems. With HTTP, for example, non-US-ASCII characters may be encoded within URLs by means of control sequences. Depending on the character set used by the end-system, such an URL may be interpreted differently (see §5.7.3).

There a different approaches to mitigate the effect of these attacks. Yet, none of them completely solves the problem. If the NIDS cannot decide which of several interpre-

[10] Ptacek and Newsham differentiate *insertion* and *evasion* attacks. Today, the term *evasion* is more commonly used to include both kinds, and we follow this terminology. We note that in [Pax99] they are called *subterfuge attacks*.

[11] RFC 791 [Pos81] *suggests* an "example reassembly procedure" which prefers the data which arrives last.

tations an endpoint will choose, in principle it could analyze *all* of them (*bifurcating analysis* [Pax99]). However, this works only in rare cases: often the number of paths to analyze grows exponentially, making bifurcating analysis computationally infeasible. In [SP03], the authors suggest to provide the NIDS with a database of per-host characteristics. The database is built offline via *active mapping*, i.e., probing local hosts to explore how they deal with ambiguous input. This approach is only feasible in small- to middle-sized networks whose topology is mostly static. It also does not work if the receiving endpoint is outside of the local network. Alternatively, the NIDS can try to derive specifics of the endpoints online by observing characteristics of their packets; e.g., by using *passive OS fingerprinting* [P0F] to guess the operating system of a host. However, such fingerprinting is based on heuristics and, hence, does not work reliably. By tweaking parameters of their local systems, users are able to mislead the fingerprinting. In [HKP01], a *normalizer* is presented which actively modifies the packet stream to remove ambiguities. While this approach is the most promising—in particular considering the recent emergence of in-line NIDS (see §2.3.6)—it may violate the end-to-end protocol semantics (e.g., diagnostic tools may cease to work). Also, it has not yet been examined whether application-layer ambiguities can also be resolved using normalization.

Denial-of-Service Attacks

A successful denial-of-service attack against a NIDS constricts the system in its intended processing. Most obviously, an attacker may try to crash (or even take control over) the system by exploiting a vulnerability. Examples of such vulnerabilities include bugs in older versions of RealSecure [CVE01b] and Snort [CVE02a]. Note that, by design, a NIDS cannot be firewalled. Both of the flaws mentioned above are located inside the systems' traffic analysis components. Therefore, they could have been exploited just by sending packets into the monitored network segment: the NIDS is examining other systems' traffic, thus there is no need to send the attack to it directly.

Another kind of denial-of-service are *resource-exhaustion attacks*. A NIDS has to get by with a limited amount of CPU time and memory. If an attacker manages to induce a larger resource demand than has been provisioned, the system will, at the very least, miss some activity. In the worst case, it will crash. A basic yet effective way to cause resource exhaustion is to expose the NIDS to a huge volume of (bogus) network traffic. Such an attack may overload the packet-capturing component, resulting in packet losses, or it may clutter up the NIDS's state tables, exhausting memory. In fact, such an attack does not need to be aimed at the NIDS itself: a flood of requests to a local host may bring down the NIDS merely as a side-effect.

However, attackers often lack the resources required to generate such amounts of traffic. Yet, even with limited resources, smart attacks may leverage weaknesses of the NIDS to achieve similar results. In [CW03], a *complexity attack* on the Bro NIDS is presented which, using a 16 Kbps data stream, causes the system to drop more than 70% of the packets. Tools like *Snot* [Sno] or *Stick* [Sti] craft packets to match known attack signatures, thereby stressing the NIDS's logging system. Even if the system is able to cope with extensive logging, the administrator will be hopelessly lost when examining the log files. "Administrator capacity" is also a limited resource.

2.4.4 State Expiration Heuristics

All major NIDSs are stateful. To avoid the state accumulating over time, filling up the memory eventually, a stateful NIDS needs to expire old state regularly. However, it is often hard to decide *when* state can be safely evicted. Consider for example connection state: if for a TCP connection, a regular four-way tear-down has been seen, it can be safely assumed that the connection has been closed.[12] Unfortunately, sometimes a NIDS does *not* see such a tear-down even when both end-points do consider the connection closed (see §3.3.3). In such cases, the NIDS does not have any clue when it is safe to remove the connection's state. For other types of state, there are similar difficulties.

In general, a NIDS has to rely on a set of heuristics for state expiration. Common approaches include time-outs, fixed size buffers, and checkpointing (i.e., regularly restarting the system to flush old state [Pax99]). [LCT+02] discusses an approach to adapt limits dynamically during run-time. More fundamentally, [LPV04] examines which kind of attacks *require* per-flow state for their detection. In §4, we discuss state expiration in more detail.

2.4.5 Evaluation Difficulties

A notorious hard problem is to evaluate the effectiveness of a NIDS. While *everybody* involved with network intrusion detection—researchers, vendors, and customers—seeks for objective quality measures, it is still unsolved. There are two main aspects of a NIDS's performance: the *quality* of the alerts, i.e., the rates of false positives/negatives; and the *efficiency* of the processing, i.e., the network volume a system is able to cope with. Unfortunately, none of them can be condensed into a single number.

The most basic problem of a NIDS evaluation is the choice of input data. Tests can be conducted either on *synthetic* traffic generated by workload generators or simulation, or on *real-world* traffic from an operational network. Both kinds of tests may either be performed *online* with live traffic or *offline* with captured packet traces. With synthetic traffic, its easy to vary its parameters (such as embodied attacks), yet exceedingly difficult to capture the complex characteristics of real-world traffic (see §3.3.3). On the other hand, real-world traffic is very specific to the environment in which it has been captured and hard to vary [RDFS04]. A common evaluation approach combines real-world with synthetic traffic: crafted attacks are injected into real-world background traffic (which is supposed to be free of attacks; an assumption that may be hard to justify). Online and offline analysis can be combined by replaying packet traces inside a test-bed network, thereby imposing a reproducible live-load on the NIDS. However, when using PC commodity hardware, significant care is required to achieve sufficient throughput and timing accuracy [FGB+03].

With respect to alert quality, the main question is how to count false positives and negatives. If, for a given input, one knows all embodied attacks, counting seems to be straight-forward: false positives are alerts which are not related to any attack; false negatives are attacks which have not triggered any alert. However, such counts are not comparable across NIDSs. Different NIDSs use different *units of analysis* [McH00], e.g.,

[12] This is simplified. In fact, with TCP there *is* reason to keep the state a bit longer.

some of them perform their analysis on a per-packet basis, others on a per-session basis. Moreover, a system may generate more than one (false) alert per (suspected) attack, and it may aggregate low-level alerts into high-level alerts. Hence, the number of alerts that NIDSs generate are intrinsically different.

To compare false positives/negatives *rates*, one needs a common denominator which is hard to find. One approach is to compare the percentage of detected attacks (i.e., attacks for which at least one alert was raised), the *coverage*. However, there are more problems. First, what constitutes *one* attack? If a scan is followed by an exploit, is this one or two attacks? If one attack, does a NIDS that only reports the scan, detect it? If two attacks, where is the boundary between attacks? Second, even assuming that we know the coverage, it is only valid for the examined set of attacks. It does not give us any insight into the system's performance on other (potentially yet unpublished) attacks.

With respect to processing efficiency, it is just as difficult to find a reasonable measure. Commercial vendors often specify the performance in packets per second that their system is able to cope with. Unfortunately, such values are rather meaningless.[13] Network traffic is very diverse. As NIDSs differ in their depth of analysis, processing time depends primarily on the traffic *mix* rather than on the total number of packets. Naturally, a system which analyzes more packets to a deeper level needs more time (see §3.3.3 and §6.3.2). In particular, many NIDS do only examine a *subset* of traffic (e.g., packets on certain well-known ports). [Ran01] summarizes the packets-per-second problem aptly: *"Measuring packets/second for IDS is like measuring miles/hour for food: some foods are moving and others aren't but it's almost always not coupled to [...] nutritional value."*.

In the literature, one of the most comprehensive evaluations of NIDSs was funded by DARPA and is presented in [LFG+00, LHF+00]; [HRLC01] further extends the evaluation method by providing a user-friendly environment called LARIAT on the one hand, and new characterizations of attack traffic on the other hand. [McH00] offers a critique of the methodology used in the DARPA studies. More recently, [DM02] evaluates several commercial systems, emphasizing the view of an analyst who receives the alerts. They find that these systems ignore relevant information about the context of the alerts. [HW02] discusses developing a benchmark for NIDSs, measuring their capacity with a representative traffic mix. In §6.3.2 we discuss our experiences with the difficulty of finding "representative" traces.

Often, for researchers it is hard to get access to realistic data. The traces captured during the DARPA evaluations have been made publicly available. Therefore, these traces have become a de-facto standard to evaluate new NIDS techniques. However, this turns out to be problematic: these traces are already well-understood and researchers may be tempted to tune their methods to this particular input. At a recent security conference, a researcher noted that he would unconditionally refuse any future paper that evaluates its work *exclusively* with the DARPA data set. While intentionally exaggerated, his point is valid: detecting well-known attacks in a well-known data set is not that hard.

[13] In fact, in personal communication several employees of NIDS vendors agreed to this claim—sometimes after presenting very optimistic packets/second rates first.

2.4.6 User Privacy

A NIDS is a surveillance device. By design, it examines—and potentially logs—the activity of *all* users which utilize a network segment. Naturally, such a monitoring poses major ethical and legal constraints whose discussion is out of this thesis' scope. However, we observe that the trade-off between user privacy and detection performance is a (non-technical) part of the site policy. For example, a site may either encourage its users to use encryption to prevent eavesdropping; or it may discourage encryption to allow a NIDS to inspect payload.

In the literature, some privacy issues are addressed. In [Joh01, Joh02], legal aspects are discussed (however, legal constraints differ by country). In [BF00], a log-file pseudonymization scheme is presented which can be reversed when the need arises (and is authorized). In [PP03], an architecture for packet trace anonymization is developed. [PLS04] discusses a privacy-preserving scheme for Internet-scale alert sharing.

2.5 Selection of Network Intrusion Detection Systems

There is wide range of NIDSs available, and giving a comprehensive overview is out of the scope of this thesis. In [Axe99a], there is a survey of research systems. Several commercial systems are compared in [Jac99]. Unfortunately, both of these summaries are rather out-dated today. Commercial NIDSs are regularly evaluated by various organizations and magazines; in particular the studies regularly performed by *The NSS Group* [NSS] receive attention. However, *The NSS Group* does not make their reports freely available. In the following, we discuss a small subset of systems:

Snort Probably the most-widely deployed NIDS. Snort is the de-facto standard among open-source systems.

Bro A very flexible open-source research system. It provides the starting point for much of our work. Hence, we take a close look at its design.

Emerald, STAT Two major research systems. They combine different detection approaches in unified frameworks. While not purely network-based, they both contain major network-based components. While STAT is open-source, Emerald is not freely available.

Dragon, IntruShield Two commercial NIDSs developed by *Enterasys Networks Inc.* and *McAfee Inc.* respectively. We have deployed evaluation versions of these systems for a couple of weeks in two of our environments.

All but the IntruShield system run on commodity PC hardware.

2.5.1 Snort

Snort [Roe99] is one of the very first—and still the predominant—open-source NIDSs. Snort was designed by Martin Roesch, and it is now primarily developed by his company *Sourcefire Inc.*. Sourcefire sells appliances based on Snort.[14] The current development version of Snort is 2.3.

Snort is primarily a misuse-based system, utilizing a large set of signatures (>3,000 currently). It runs on commodity hardware, leveraging the freely available library *libpcap* [Lib] for portable packet capturing on Unix and Windows platforms. By itself, it does not provide any graphical user interface. However, given some general system administration skills, Snort is easy to deploy due to its rather low complexity. Its configuration, including the signatures, is stored in ASCII files. Similarly, the primary logging facilities are also ASCII-based; yet, an interface to database systems is provided.

Architecturally, Snort consists of a core extended with a set of plug-ins which implement most of the more sophisticated functionality. Its main unit of analysis is a packet. Each packet observed on the network is first passed through a set of *preprocessors* which may extract information and/or modify the packet. Then *detection plug-ins* match the packet against signature conditions. If a match is found, *output plug-ins* notify the administrator.

Besides signature matching, Snort contains additional detection mechanisms, including a port-scan detector and reporting of protocol violations.[15] An earlier version of Snort included a statistical anomaly detection engine [SHM02]. Apparently, there are efforts to revitalize this project.

Snort does not include any communication sub-system. Therefore, it can only be used as a stand-alone system. If a network is to be monitored with Snort at multiple points, the results of all instances need to be correlated externally.

2.5.2 Bro

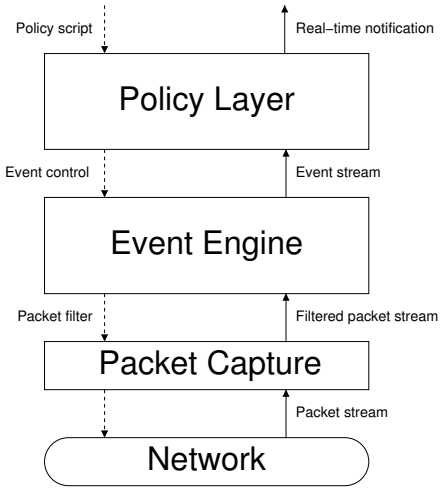
Bro [Pax99] is one of the most flexible open-source NIDSs. It is designed and primarily developed by Vern Paxson. In contrast to most NIDSs, it is fundamentally neither an anomaly-based system nor a misuse-based system. Rather, its core is *policy-neutral*. Bro is written in C++ and covered by a BSD-style license.

Bro's primary design goals were *(i)* separation of mechanism and policy, *(ii)* efficient operation suitable for high-volume network monitoring, and *(iii)* resistance to attacks directed at itself. To this end, Bro's architecture consists of three layers: packet filtering, event generation, and policy script execution (see Figure 2.3). With increasing layer, the volume of processed data decreases, thereby enabling more expensive processing. Packet filtering is done using a static *BPF* expression [MJ93], leveraging *libpcap* like Snort. The *event engine* generates *events* which—in the sense as defined in §2.3.1—represent policy-neutral abstractions of network activity at different semantic levels. For example, there are events for attempted/established/terminated/rejected connections,

[14] In this thesis, we only consider the freely available version.

[15] Such violations are sometimes called protocol *anomalies*. However, this terminology does not match well with its common meaning in anomaly detection. Therefore, we avoid the term.

Figure 2.3 Architecture of the Bro system (adapted from [Pax99]).



the requests and replies for a number of applications, and successful and unsuccessful user authentication. The user writes *policy scripts* using a specialized, richly-typed high-level language. These scripts contain *event handlers* which are executed when the corresponding event is raised. Event handlers codify the actions the NIDS should take: updating data structures describing the activity seen on the network, sending out real-time alerts, recording activity transcripts to files, and executing programs as a means of reactive response.

Bro's main unit of analysis is a connection. While for TCP the definition of a connection is straight-forward, the system also fits UDP and ICMP into its connection model by using a flow-like definition. Connection semantics are interpreted by *analyzers* which follow the endpoint's communication, extracting basic semantic protocol elements and generating corresponding events. Usually, an analyzer consists of two components: one inside the event engine which performs policy-neutral analysis and generates events; and a second component in the form of a policy script containing (predefined yet customizable) policy-specific actions. Analyzers are available for a wide range of transport- and application-layer protocols, including TCP, HTTP, FTP, SMTP, SSL, and DNS. Moreover, protocol-independent analyzers detect scanners [JPBB04], stepping stones [ZP00b], backdoors [ZP00a], etc. A generic connection analyzer generates one-line ASCII summaries of any connection the system sees. To reduce the processing load, Bro examines only packets requested by at least one of the analyzers; others are filtered at the lowest layer by installing a suitable filter. For instance, the connection analyzer requests only TCP control packets (i.e., SYNs/FINs/RESETs) rather than all TCP packets; these are sufficient to deduce basic connection characteristics such as duration and size of the transfered payload.

As noted above, Bro was designed to resist attacks against itself. One consequence of this design guideline is the need to avoid predictability; e.g., while the systems needs to expire old state, an adversary should not be able to predict *when* it does it. To

achieve this, Bro's design assumes that an attacker is not aware of the system's concrete parametrization. Therefore, the user needs to adapt the default policy scripts shipped with Bro before using them. This is a variant of Kerkhoff's principle: while the detection mechanisms are public (Bro is open-source), their parameterizations are not. In general, during Bro's development, attack resilience always had priority. In §4.4, we discuss the consequences for Bro's deployment in high-performance networks.

In the past, Bro has been a non-distributed NIDS. Yet, as our work has progressed, it has turned into a fully distributed system, capable of coordinating physically separated instances. Moreover, it is not anymore restricted to network input, although we expect that network-tapping will remain its primary mode of operation. In §5, we discuss our extensions in this context.

In summary, Bro is a policy-neutral research system which provides a large degree of flexibility to experiment with different approaches to network intrusion detection. Therefore, we carried out much of our work in the context of this system. When problems turned up, the open-source license allowed us to understand them quickly and, if required, fix them. All of our modifications are now included in Bro's distribution.

2.5.3 Emerald

Emerald [PN97] is an intrusion detection framework developed by *SRI International*. Its primary target environment is a large-scale heterogeneous enterprise network, consisting of several independent sub-units with differing trust relationships. Emerald is not freely available.

Emerald is a highly-distributed system that uses a 3-layer architecture: the *service analysis* layer monitors individual hosts and network components, the *domain-wide analysis* layer correlates results across a sub-unit's service layer, and the *enterprise-wide analysis* layer coordinates several sub-units. On all three layers, *monitors* make up the basic building blocks. Each monitor contains modules for data acquisition, detection, and correlation. All monitors are built from the same code-base; *resource objects* encapsulate input specifics. On the service layer, a monitor acquires its input from the monitored components. On higher layers, it communicates with other monitors. The communication between monitors uses a subscription-based scheme to exchange asynchronous messages. The communication model provides both "push" and "pull" semantics. Inside a monitor, several detection components can be used which include an anomaly detector (based on *NIDES* [JV93]), and a misuse detector (based on the expert system *P-BEST* [LP99]).

In general, an Emerald set-up may include both host- and network-based service-level monitors. With respect to the classical categorization, it is therefore a hybrid. The application of Emerald's detection modules to network traffic is discussed in [PV98] in more detail.

2.5.4 STAT

The *STAT tool suite* [VEK00] is an intrusion detection framework developed by the University of California, Santa Barbara. It is a distributed misuse-based system that models attack scenarios with state transition diagrams. STAT is open-source under the terms of the GNU General Public License [GPL].

An attack scenario consists of states and transitions. The states represent snapshots of a system's security relevant characteristics. Transitions from one state to another are triggered by actions which represent steps of an attack. If there is a series of transitions leading from an initial starting state to a "compromised" ending state, a successful attack has been detected.

Initially, this state-transition approach was used independently for host- and network-based detection (*USTAT* [Ilg93] and *NetSTAT* [VK99], respectively). Now STAT provides an unified framework for both systems in which attack scenarios are defined in a custom language called *STATL* [EVK02]. MetaSTAT [VKB01] adds dynamic reconfiguration capabilities to the STAT framework.

The network-based module of STAT, *NetSTAT*, consists of four types of components: the *network fact base* stores all security relevant network information; the *scenario database* contains state transition diagrams; *probes* are installed at points of interest across the network to detect attacks; and an *analyzer* pre-calculates the probes' configurations.

2.5.5 Enterasys Dragon

The Dragon NIDS [Dra] is developed by *Enterasys Networks Inc.*. It runs on standard PC platforms, and is available both as a software-only package and as a boxed appliance in different variants.

Dragon is a primarily signature-based NIDS. However, it also contains a set of other heuristics such as scan detection (by counting connection attempts) and ICMP covert channel detection (by noticing unsolicited ICMP replies). It is a non-inline system which can execute active response by injecting packets into the network. Conceptually, it consists of two different components. *Sensors* are Dragon's main detection units, performing packet capture and analysis. They connect to a *management server* which combines the results of all sensors. Such a management server can also include host-based Dragon sensors into its analysis. For configuration, the Dragon system provides a stand-alone Java-client which connects to the management server. For data analysis, the management server provides an HTTP interface for browser access. Additionally, both sensors and manager allow SSH logins.

Dragon contains components for analyzing protocols such as FTP, DNS, HTTP, and SMB. Furthermore, TCP streams are reassembled as configured. A range of filters and specific logging requests can be configured. While Dragon's signature set is not publicly available, it is open in the sense that when having a license you can examine the signatures; they are coded in XML.

In §4 we discuss our experiences with running an evaluation setup of Dragon 7.01 for a couple of weeks in one of our environments (see §3.2.1). It consisted of two appliances:

INS2 Integrated Network Sensor/Server. A box providing both a network sensor and a management server. It contained a dual-Xeon 2.4 Ghz with 1 GB RAM, 33 GB hard drive, 3 Intel Pro/1000 network interface cards (one fiber, two copper), and runs Linux 2.6.[16]

GE250 Dragon Network Sensor Appliance. An additional sensor, requiring a remote manager for operation. It contained the same hardware than the INS2 yet only 512 MB of RAM.

In the following, when we talk about the Dragon's "default configuration", we are referring to the configuration as installed by the security consultant who was authorized by Enterasys to setup the system in our environment. As far as we can tell, his modifications to the initial configuration were minor.

2.5.6 McAfee IntruShield

The IntruShield NIDS [Int] has been developed by *IntruVert Networks Inc.* which has now been acquired by *McAfee Inc.*. IntruShield ships as boxed appliances using custom hardware in different variants, depending on the size of the target environment.

IntruShield can be installed in-line to provide active response; alternatively (or even at the same time), it can be deployed in traditional tap-mode. The system is primarily signature-based, yet also includes a set of other heuristic mechanisms, e.g., statistical anomaly detection (such as recognizing DOS-attacks based on thresholds) and reporting of protocol violations. The current version can decrypt SSL-encoded traffic when provided with the corresponding SSL private key. The primary interface to the system is a browser-accessible GUI, written in Java and running on a separate Windows system.

Like Dragon, IntruShield provides application-layer support for a large set of protocols. Unlike Dragon, IntruShield's signature are closed, i.e., the user cannot inspect the precise conditions that led to a match. If one can't figure this out from the informal description each signature is accompanied with, one has to send the affected packet to McAfee's support.

The IntruShield appliances are based on a custom hardware platform. While they use standard Intel processors for general management, they include network processors, ASICs and FPGAs to speed-up computing intensive tasks (e.g., signature matching and SSL-decoding). Unfortunately, McAfee does not provide concrete details about the system's internals.

We had access to an IntruShield 4000 for a couple of weeks which, at that time, was the most powerful of the IntruShield appliances available. It is specified for networks up to 2 Gbps and 1,000,000 concurrent sessions, and provides four 1 Gbps-ports.

[16] Quotation from Enterasys' data sheet: 'Dragon Enterprise Management Server is made up of a number of highly integrated technologies.".

3 High-Performance Environments

> The tendency of an event to occur varies inversely with one's preparation for it.
>
> — *David Searles*

The basis of our work is our experiences with network security monitoring in environments of different characteristics. In this chapter, we first discuss the environments that we use to explore our research ideas, focusing on their network infrastructure. Then we identify the main properties of high-performance environments relevant for network intrusion detection.

3.1 Overview

Our work draws from experiences with network intrusion detection in seven environments of different characteristics. Table 3.1 summarizes basic characteristics of these networks.[1] The environments are:

Munich Scientific Network, Germany (MWN).
: Large research network providing Internet connectivity to two major universities as well as several research institutes in the Munich area.

University of California, Berkeley, CA, USA (UCB).
: Large campus network providing Internet connectivity to the university and affiliated institutes.

Lawrence Berkeley National Laboratory, CA, USA (LBNL).
: Medium-sized research network.

National Energy Research Scientific Computing Center, Oakland, CA, USA (NERSC).
: Medium-sized research network, primarily providing super-computing services.

Saarland University, Saarbrücken, Germany (USB).
: Medium-sized university network.

International Computer Science Institute, Berkeley, CA, USA (ICSI).
: Small research network; upstream connectivity provided by UCB.

Work Group Network, TU München, Germany (WGN).
: Small research network; upstream connectivity provided by MWN.

[1] In this chapter, we discuss the networks' states as of 2005. We note that, with respect to infrastructure and traffic characteristics, all of them are moving targets.

Table 3.1 Basic characteristics of the environments (estimated).

	MWN	UCB	LBNL	NERSC	USB	ICSI	WGN
Backbone	10 Gbps	2 Gbps	1 Gbps	10 Gbps	1 Gbps	1 Gbps	100 Mbps
Upstream	1 Gbps	3×2 Gbps	1 Gbps	2.5 Gbps	155 Mbps	100 Mbps	1 Gbps
Total/month	55 TB	130 TB	35 TB	66 TB	10 TB	430 GB	150 GB
In/month	26 TB	65 TB	11 TB	55 TB	7 TB	200 GB	50 GB
Out/month	29 TB	65 TB	24 TB	11 TB	3 TB	230 GB	100 GB
Users	100,000	50,000	4,000	2,000	20,000	300	150
Hosts	50,000	45,000	13,000	2,000	10,000	200	80
DNS entries	30,000	80,000	13,000	3,500	20,000	300	200

The primary mission of these environments is supporting research. This focus distinguishes them from commercial enterprise networks which usually have different security demands and face different threats.

In this thesis, we refer to networks featuring Gbps uplinks which transfer tens of terabytes a month as *high-performance* environments. For the environments discussed here, this applies to MWN, UCB, LBNL, and NERSC; Figures 3.5, 3.6, 3.7, and 3.8 show 7-day snapshots of their TCP application mixes (we discuss them in §3.3.3). Moreover, we refer to networks which primarily provide network connectivity to independently administered LANs as *backbone* networks. On the other hand, we term environments which are (mostly) centrally managed as *stub* networks. Using this terminology, MWN, UCB, and USB are backbone networks while LBNL, NERSC, ICSI, and WGN are stub networks.

After briefly presenting the seven environments in §3.2, we discuss their characteristics in terms of policies (§3.3.1), threats (§3.3.2), and traffic (§3.3.3).

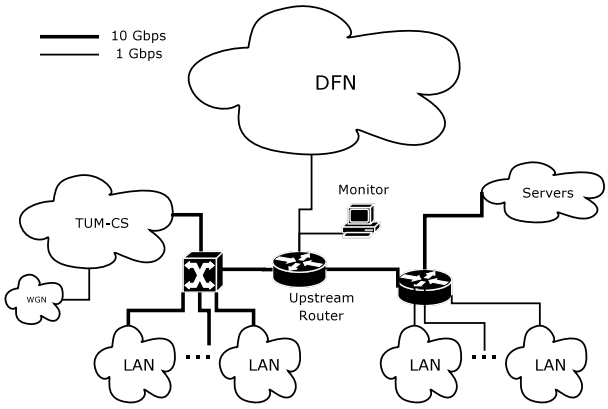
3.2 Environments

We now summarize properties of the seven environments, addressing network infrastructure, traffic mix, security monitoring, and firewall policy (all of the environments deploy a firewall). We note that sometimes we need to be deliberately vague as some of the information is considered sensitive. With respect to upstream connectivity, we only discuss the primary links, ignoring backup links which are normally not in use.

3.2.1 Munich Scientific Network

Our primary research environment is the *Münchner Wissenschaftsnetz* (Munich Scientific Network, MWN, [MWNa, MWNb]). MWN provides Internet connectivity to most non-commercial research facilities located in the Munich area, Germany, including the two universities *Technische Universität München (TUM)* and *Ludwig-Maximilians-Universität München (LMU)* as well as several Max-Planck- and Fraunhofer-Institutes. MWN is operated by the *Leibniz-Rechenzentrum* (Leibniz Computing Center, LRZ) of the *Bayerische Akademie der Wissenschaften* (Bavarian Academy of Sciences and Humanities).

Figure 3.1 Backbone topology of MWN (adapted from [MWNa]).



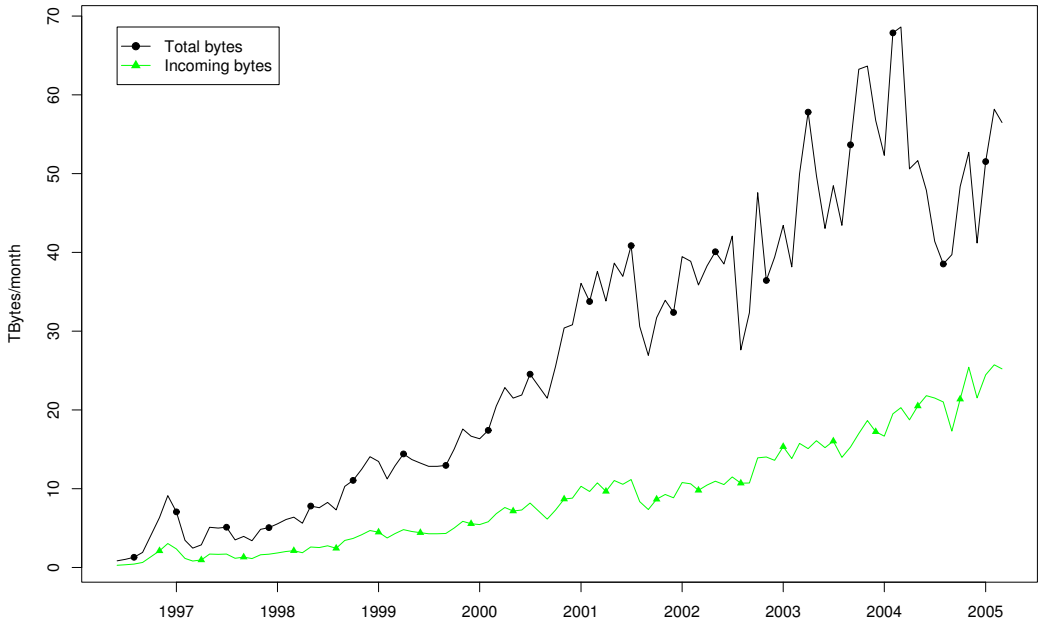
In Germany, MWN is one of the largest research networks. Figure 3.1 shows an overview of the network's topology. MWN deploys a 10 Gbps backbone which interconnects more than 200 individual LANs, distributed across an area of 60 km in diameter. LAN connectivities range from 10 Gbps links to 64 Kbps ISDN lines. MWN encompasses 50,000 individual hosts (30,000 of them with registered DNS names) and 100,000 registered users of which two thirds are students. The LRZ also provides a wide variety of network-based services to its customers, such as mail relays (1.2 M mails/day), remote backups (2 TB/day), virtual Web servers, proxies, VPN servers, and super computers.

MWN's upstream is a 1 Gbps link to the *Deutsches Forschungsnetzwerk* (German Research Network, DFN). On a monthly average, external traffic encompasses 26 TB of incoming and 29 TB of outgoing data. Figure 3.2 shows the historic development of MWN's external traffic. We see that the external traffic has increased from 850 GB per month in 1996 to 55 TB per month in 2005, with a peak of 70 TB in 2004, even though the major peer-to-peer applications are blocked at the perimeter (see below). For the future, an increasing trend is expected to hold. The average link utilization is 84/93 Mbps, with peaks reaching 400 Mbps. Most of the external traffic is HTTP (50–60% of the total bytes as well as of the total TCP connections). The network includes a set of popular hosts in the `leo.org`-domain, located at TUM and operated by the computer science's system administration group. These hosts account for a major share of the traffic: `ftp.leo.org` (10 TB/month) is an FTP server mirroring a large set of free software[2]; `dict.leo.org` (10–20% of all TCP connections, 7-8 TB per month) is an HTTP server providing a German-English dictionary; `irc.leo.org` is one of the major German IRC servers.

All MWN routers—in particular, the upstream router (see Figure 3.1)—are Cisco 6509s. Additionally, there are hundreds of HP/3COM switches deployed. To accom-

[2]The server is currently being reorganized. Therefore, there are regular down-times and a reduced set of available software—which is the reason for the decrease of MWN's overall traffic since 2004 that we see in Figure 3.2.

Figure 3.2 History of MWN's traffic volume.



The data for this plot has been generously provided by Alfred Läpple of the LRZ.

modate future demands, the DFN uplink will be upgraded to 10 Gbps in 2006. Also in 2006, the backbone's topology is going to change to a triangular structure to increase redundancy.

The DFN uplink is MWN's main point to monitor and enforce its firewall policy. Most of the external traffic is subject to a set of restrictions which are enforced by blocking certain TCP/UDP ports. Table 3.2 shows a subsets of ports being blocked. In particular, many ports known to be used by peer-to-peer file-sharing systems are closed. This is due to both the potentially large volume of such applications as well as their legal implications. Moreover, filters are installed to prevent spoofing and broadcast-pings.

The LRZ relies on custom software for monitoring the uplink, generating activity summaries in real-time. Of particular interests are high-volume hosts, scanners, and SMTP traffic. A Snort system is also deployed using a very small set of custom signatures. Currently, several commercial NIDSs are being evaluated for their utility in the MWN environment.

For our work, we had access to the full traffic of MWN's upstream link. On the upstream router, a monitor port is configured to copy all packets to a dedicated Gbps Ethernet link. This link is connected to a passive optical splitter which duplicates the traffic to four individual monitoring machines. These monitors are mostly built from commodity hardware and run open-source operating systems; Table 3.3 summarizes the hardware. One of the machine features a high-performance *Endace DAG* monitoring card which provides reliable packet capturing for Gbps links. We discuss the use of this card for network intrusion detection in §4.5.4. The four monitors are connected by a

Table 3.2 Subset of ports blocked at MWN's perimeter [MWNc].

Ports	Service	Incoming	Outgoing
25	SMTP	•	
53	DNS		•
111	RPC	•	•
161, 162	SNMP	•	•
135, 137-139, 445, 593	Windows Networking	•	•
515, 613	Printing	•	•
512-514	Remote Access	•	•
1214, 4661-4665, 6346, 6347, 6699	File Sharers	•	•
1433, 1434	Windows SQL-Server	•	•
1025, 1034, 2745, 2766, 3127, 5000, 5554, 6129, 9898	Worms, Backdoors, Trojans	•	•

100 Mbps switch. For some of the evaluations presented in subsequent chapters, we used additional machines.

We note that, in general, converting traffic from a bi-directional 1 Gbps link into a unidirectional 1 Gbps link may exceed the available bandwidth. We discuss such artifacts of the monitoring environment in §4.2.6. However, the average link utilization suggests that our setup is fine for the current volume of external traffic.

3.2.2 University of California, Berkeley

The University of California at Berkeley (UCB, [UCB]) is a public university which is consistently ranked as one of the top academic institutions in the U.S.. It has about 35,000 students and a faculty/staff of 15,000.

The campus's backbone network provides access to about 45,000 hosts. The network's backbone capacity is 2 Gbps, made up of parallel, fully-redundant 1 Gbps links. UCB has three main external links, each having a capacity of 2 Gbps as well: *(i)* to commodity Internet, being rate-limited to 200 Mbps per direction; *(ii)* to CENIC's Digital California Network [DC] which serves all educational sites in California, and connects to several exchange points; *(iii)* to CENIC's High Performance Research Network [HPR], serving the major California research universities including other UCs, and also connects to some exchange points. The border routers are two Juniper M40s, one Cisco 7500, and one Cisco 12008. Internally, Cisco 6500s/7500s/7513s are used for routing. The network infrastructure is leveraged by several affiliated institutes (e.g., ICSI; see §3.2.6).

In total, there are 65 TB of traffic per month incoming as well as outgoing. On average, that corresponds to 210 Mbps in each direction. The dominant applications are HTTP and peer-to-peer software, e.g., BitTorrent and Gnutella.

The external firewall only imposes a very small set of restrictions. For outgoing traffic, there is no blocking at all. For incoming traffic, a few ports are closed, most notably SNMP and several ports primarily used by Windows, e.g., NetBIOS and Microsoft-DS.

Table 3.3 Systems monitoring MWN's upstream link.

	Monitor 1	Monitor 2	Monitor 3	Monitor 4
OS	Linux 2.6.x	Linux 2.6.x	FreeBSD 5.2.x	FreeBSD 5.2.x
CPUs	2×Opteron 1.8 Ghz	2×Xeon 3.0 Ghz	2×Opteron 1.8 Ghz	2×Xeon 3.0 Ghz
Memory	2 GB	2 GB	2 GB	2 GB
NIC	Pro/1000[a], DAG[b]	Pro/1000	Pro/1000	Pro/1000
Disk	1 TB	1 TB	1 TB	1 TB

[a]Intel Pro/1000 SX.
[b]Endace DAG 4.3GE (see §4.5.4).

The external links are monitored by the Snort, Bro and IntruShield NIDSs. For this purpose, the external traffic is merged into a single 1 Gbps link by means of an RSPAN-VLAN [RSP]. Due to its capacity limitation, such a setup introduces monitoring artifacts (see §4.2.6).

3.2.3 Lawrence Berkeley National Laboratory

The Lawrence Berkeley National Laboratory (LBNL, [LBL]) is the oldest of the U.S. Department of Energy's (DOE) national research laboratories. It is managed by the University of California. The lab conducts unclassified research across a wide range of scientific disciplines, including studies of the universe, quantitative biology, nanoscience, new energy systems and environmental solutions. The lab's network provides access to about 4,000 users and 13,000 hosts. The network is centrally managed and includes a few off-site LANs. The internal backbone is operated at 1 Gbps. Upstream connectivity is provided by a 1 Gbps link to DOE's nation-wide Energy Sciences Network (ESNet). LBNL's core switches/routers are Cisco 6500/7500s, the border router is a Force10 E1200.

There is about 35 TB of external traffic per month, 11 TB incoming and 24 TB outgoing (36/78 Mbps on average; peaks are reaching 850/600 Mbps). The average work load of the network's central core switch is 161/154 Mbps. The dominant application protocols are HTTP, SSH, and FTP-DATA.

In general, the external firewall only blocks a small number of ports, yet for certain critical hosts the rules are more strict. Using the Bro NIDS, LBNL monitors the network traffic at several locations, including the external connection and its DMZ. Active optical taps provide access to the packets. If certain attacks are detected, the originator's connectivity is automatically dropped.

3.2.4 National Energy Research Scientific Computing Center

The National Energy Research Scientific Computing Center (NERSC, [NER]) is one of the largest facilities in the world providing computational resources for scientific research. It is funded by DOE. While NERSC is administratively part of LBNL, it is physically

separate and uses a different Internet access link. It contains about 2,000 hosts and provides access to 2,000 users, with a staff of about 75 people. NERSC's internal backbone is a 10 Gbps network. The network's upstream is an OC-48 link to ESNet. The main routers are from Juniper Networks.

NERSC's external traffic is about 66 TB a month, with 5/6th of it incoming. On average, that is 178 Mbps in and 36 Mbps out. The total busy-hour utilization of the link is 260 Mbps. The traffic is dominated by large file transfers; the major applications are FTP-DATA, HSI (an interface to NERSC's *High Performance Storage System [HPP]*), and SSH.

NERSC's external firewall only blocks a small subset of ports by default. NERSC monitors its network traffic at the border as well as at internal core routers, using the Bro and Snort NIDSs on FreeBSD systems. Originators of certain attacks are automatically dropped.

3.2.5 Saarland University

The Saarland University (USB, [USB]) is a medium-sized university with 15,500 students and 2,000 employees, covering a broad range of areas in research and education.

The university's main campus has about 10,000 hosts and is equipped with a 1 Gbps backbone which has a 155 Mbps upstream to the DFN. There are about 7 TB outgoing and 3 TB incoming traffic per month (23/10 Mbps on average), with HTTP being the dominant application-layer protocol. In addition, several affiliated institutes leverage the infrastructure. In total there are more than 400 VLANs. A Cisco 6500 is the main internal switch; the upstream is managed by two redundant Cisco 7200s.

The firewall policy differentiates between incoming and outgoing traffic. For incoming connection, only selected service/host pairs are permitted as destinations. For outgoing connections, only a small set of restrictions apply. (For individual LANs customized policies may be provided). All internal clients need to be authorized, e.g., by using a VPN gateway. The upstream link is operationally monitored with IP- and NetFlow-based accounting. Misbehaving hosts are detected and blocked manually.

3.2.6 International Computer Science Institute

The International Computer Science Institute (ICSI, [ICS]) is an independent, nonprofit research institute affiliated with UCB. ICSI's research focuses on algorithms, artificial intelligence, networking, and speech processing.

ICSI is connected to the Internet by a 100 Mbps link to UCB. Internally, a few central systems have Gbps connectivity while most of the around 200 hosts use 100 Mbps links. There are about 300 users with 100 of them on-site. The two redundant border routers are Cisco 7206s. During a month, there is about 200 GB of external traffic inbound and 230 GB outbound (on average 80/100 Kbps). The dominating applications are HTTP and SSH. ICSI's firewall closes a selected subset of ports. External traffic is monitored by the Bro NIDS running on a FreeBSD host.

3.2.7 Work Group Network at TU München

The smallest environment considered in this thesis is that of the Network Architecture group (WGN, [NET]) at TUM. It hosts staff workstations, student pools as well as labs. The group's research focus is Internet technology.

Ignoring lab devices, the network's capacity is 100 Mbps. The network is connected by a 1 Gbps link to the computer science department of TUM. This department is in turn part of MWN which hence provides the upstream connectivity. There are 50/100 GB of incoming/outgoing traffic a month. A HP ProCurve 2524 switch is the central network component. It also provides a monitor port for the network's security monitoring performed by the Bro NIDS running on a Linux system. The firewall policy is strict: in general, for incoming traffic only SSH is permitted. Internally, there are a few subnets which do not have any direct Internet connectivity.

3.3 Characteristics of High-Performance Environments

Despite their differences, the environments discussed in §3.2 show interesting similarities. We now discuss their main characteristics relevant to network security monitoring. We believe that many of our observations apply in general and thus to other environments as well.

In §3.3.1 we first discuss aspects of the network's policies. Then, in §3.3.2, we examine typical threats which they face. Finally, in §3.3.3 we discuss traffic characteristics.

3.3.1 Policies

In all of our environments, the site policy is primarily specified by "terms of use" (TOU) and similar regulations that users need to accept when getting access. The terms specify what the intended (and accepted) use of the network is. Often, multiple TOUs apply; e.g., for WGN users, there are the TOUs of TUM's computer science department, the MWN, and the DFN.

In all of these environments, the TOUs (and thus the site policies) are quite liberal, giving the users a large degree of freedom. This is due to their focus on research: for researchers and students alike, the network is one of the most important tools; it is part of the infrastructure as well as a subject of research itself. Moreover, the research community tends to be an early adopter of new network technologies. This requires a liberal policy with regards to permitted network usage. Consequently, the number of concrete restrictions tends to be rather small; experience indicates that any constraint imposed by the policy is going to hurt some researcher.

Most importantly, the TOUs require users to avoid any "inappropriate use" and forbid giving access to other persons (see, e.g., [MWNd] for MWN). The networks use two main methods to verify policy compliance: *(i)* certain rules are enforced by firewalls, and *(ii)* the networks are continuously monitored to detect violations.

When installing a firewall, there are two general approaches to set it up: a network may favor openness, posing restrictions only to a chosen set of services (*default-open*); or

it may enforce the more strict rule of denying everything which is not explicitly allowed (*default-close*). In accordance with the liberal site policies, the default-open approach is the prevalent one among our environments. The strategies to handle firewall restrictions vary; the three backbone networks all follow different approaches. UCB is very reluctant to impose *any* restrictions at all. At MWN, there is a default policy which imposes a well-defined (and published) set of constraints, such as services which cannot be used externally. However, variants of the policy are available to individual subnets. If subnets require less restrictions or prefer to handle their own policy, this is usually possible. For example, WGN includes a subnet used for honeypot research which is completely relieved from any external restrictions. On the other hand, if a subnet prefers a more strict policy, this can also be provided, potentially by leveraging services offered (and secured) by the LRZ. At USB, for outgoing traffic only a small set of applications is disabled. However, for incoming connections, there is a strict default-close strategy in place. In general, even incoming SSH sessions are denied. If a service is to be provided to external users, this needs to be specifically requested.

The environments also monitor the networks for policy violations. However, a liberal informal site policy limits the applicability of a NIDS. While "inappropriate use" is forbidden, there is no precise definition which could be verified automatically.[3] Therefore, the environments look for activity which, based on their experience, *usually* tends to be malicious, such as inconspicuous hosts suddenly transferring high volumes of data, or FTP servers on non-standard ports. By deploying such a monitoring strategy, the network operators effectively *imply* concrete policy rules (e.g., a host may not run an FTP server on a non-standard port). We term the set of such implied rules an *experience-based policy*. At MWN, the experience-based policy is primarily monitored with custom software as most NIDSs lack the required flexibility.

An interesting observation about experience-based policies is that they tend to make statements about the *consequences* of an attack rather than the attack itself. Most hosts are compromised to exploit their resources, e.g., to share files. Such resource misuse is easier to detect than the initial exploit since typically it changes the traffic pattern of the victim significantly. Similarly, a host infected by a worm is often recognizable by its attempts to spread further.

Experience-based detection applies primarily to *internal* hosts. This is due to three reasons:

1. The objective is to protect internal hosts.

2. For internal systems, it is possible to contact the administrator in charge. When detecting a compromised host, a backbone network's main tool to avoid further harm is to drop the host's connectivity. However, observed symptoms may be due to either malicious or benign activity. If it is in fact benign, dropping may unnecessarily impact operation.

[3] The exceptions are a few concrete rules which are part of some TOUs; MWN's TOU, for instance, requires users to avoid passwords which are easy to guess.

3. Tracking external victims is impractical. Due to the large number of misbehaving external systems, it is not feasible to contact all the persons in charge (who are often hard to find anyway). Even if one does contact some of them, they might prove unresponsive.

Site policies can also contain rules specifying *reactions* to detected violations. One very interesting aspect of LBNL/NERSC's site policies is their active response [JPBB04]. If one of their Bro NIDSs identifies an external host scanning the network, the originating IP address is automatically blocked on the ingress router for a certain amount of time. Many network administrators are reluctant to deploy such automatic drops of connectivity; they are afraid of denial-of-service attacks which manage to exploit the response system to deny benign traffic. Yet, the experience of these sites shows that the number of compromised hosts significantly decreases once blocking is used. In the case that an address has been wrongfully dropped, connectivity can be quickly restored manually once it has been diagnosed.

3.3.2 Threats

For the backbone networks, large-scale *undirected* attacks are the major threat. In personal communication, we often heard statements like *"we are looking for the* big *guys [in terms of volume]; we do not have much of a chance to find an individual well-directed attack anyway."* Put another way, the backbones' underlying assumption is that the administrators of connected LANs take care of their own systems. The backbones are mainly concerned about attacks sufficiently large-scale that they affect *their* operation and, thus, impact all users of the backbone. This is a rather pragmatic yet practicable approach.

In terms of undirected attacks, there are two main threats:

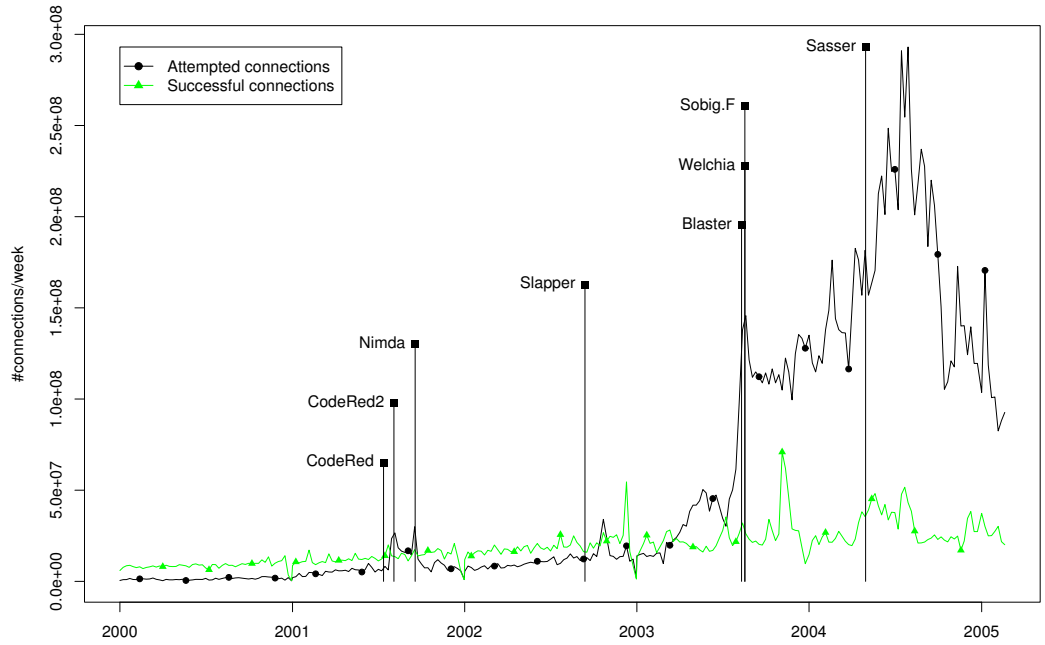
Misuse of Resources. Once a host is compromised, typical forms of misuse include illegal file-sharing, sending spam, and participating in distributed denial-of-service attacks. Networks such as MWN and UCB are preferred targets: their Gbps uplinks to high-performance research networks are among the most powerful Internet connections available. For many attackers, it is very tempting to get control over hosts inside these networks to leverage their connectivity. However, many automated attacks do not consider location; they simply strive to compromise as many hosts as possible.

Worms. A worm has the potential to infect thousands of hosts within minutes.[4] In addition to the damage caused on individual machines, the network-load triggered by a worm can be immense [MPS+03].

Any public network is constantly under attack [PYB+04]. The number of undirected attacks has drastically increased over the last years. Nowadays all systems connected

[4] In fact, theoretically it has been shown that a worm could infect 95% of one million vulnerable hosts in 510 milliseconds [SMPW04].

Figure 3.3 History of connection attempts seen at LBNL's perimeter.

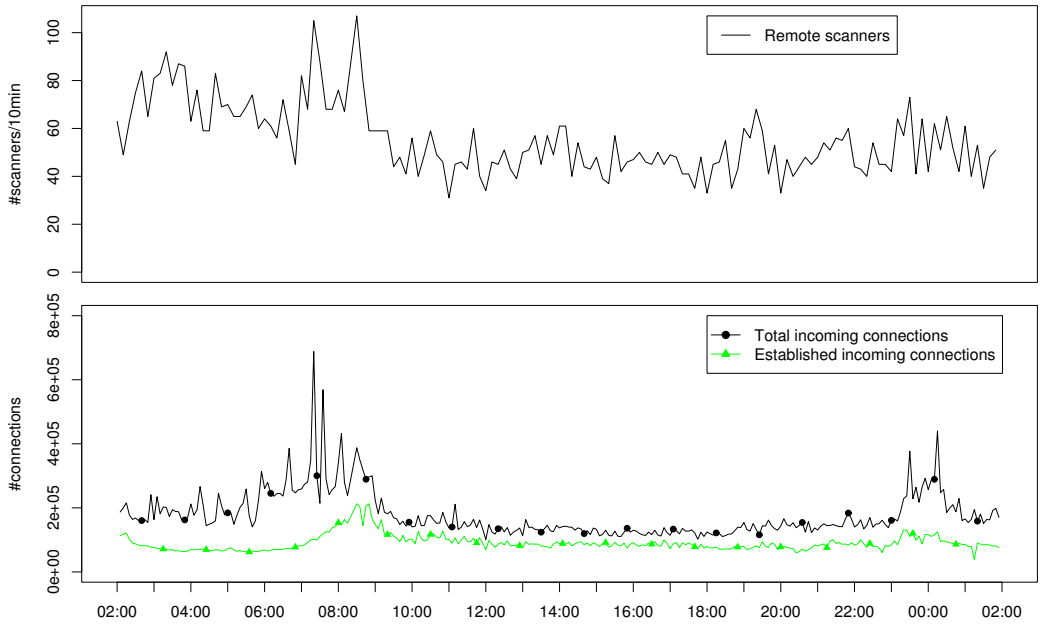


Connections are classified as non-successful when an initial SYN either produced no response at all or was answered with a RESET. All other connections are assumed to be successful. A similar plot has originally been devised by Mark Dedlow of LBNL who has also generously provided the connection data.

to the Internet are regularly probed for vulnerabilities, often a couple of times an hour. Figure 3.3 shows the increase of failed TCP connection attempts into LBNL's network since 2000. In January 2000, on average only 11% of all connection attempts did not get established. Five years later, this is up to 80%, with peaks reaching more than 90%. The vast majority of these connection attempts is due to Internet-scale scans, often seen after worm outbreaks. For MWN, Figure 3.4 shows that a large fraction of such failed connection attempts are indeed due to scans. Figure 3.4(top) shows the number of such attempts per 10-minute interval for a day's worth of traffic at MWN. For the same time interval, Figure 3.4(bottom) shows the number of identified scanners reported by the Bro NIDS. We see that the number of scanners closely correspond to the number of connection attempts. In total, during that day we saw 7,910 distinctive scanning sources. About 50% of all connection attempts failed.

There are also directed attacks that are sufficiently significant to impact the operation of a backbone network. In particular, distributed denial-of-service attacks targeting a local host may be noticeable. While the prevalence of such attacks is hard to quantify, a study in 2001 observed more than 12,000 world-wide denial-of-service attacks within a period of three weeks [MVS01]; [MDDR05] concludes that denial-of-service attacks are *"extremely common ... [and] there is reason to believe they will become even more popular"*. Such attacks have not been a major issue at UCB yet. However, MWN includes

Figure 3.4 Scanning activity seen at MWN's perimeter.



Scanners are sources performing at least 50 distinctive TCP/UDP connection attempts. Each source address is only counted the first time it is encountered. Note that many sources perform multiple scans, regularly involving thousands of individual connection attempts.

the IRC server `irc.leo.org` which—like any major IRC server—is a regular victim of large-scale flooding attacks. Nevertheless, MWN's capacity has always been sufficient to avoid impacting the performance of the rest of the network.

Due to their central administration, stub networks such as LBNL and NERSC are able to put more efforts into detecting directed attacks which do not change a host's traffic pattern significantly. In the DOE environments, account compromises are of particular concern; NERSC, e.g., centrally analyzes syslog messages of all SSH servers. At WGN we verify the network's firewall with an independently implemented Bro configuration, mimicking the firewall rules. As the network is rather small, we can track down individual unexpected connections. Such a strategy would be infeasible at MWN-scale.

Finally, we note that while by definition a threat is a "potential policy violation" (see §2.2.3), in practice threats often impact the concrete site policy. For instance, a newly released worm may lead to a policy rule which forbids the use of the exploited service. Thus, there is in fact a circular dependency between policy and threats.

3.3.3 Traffic Diversity

Large networks exhibit an immense diversity in their traffic, both when seen *between* different environments as well as *within* a single network. One of the main characteristics is an environment's *application-mix*, i.e., the fractions that the major network applications

contribute to the total volume. For NIDSs, the application-mix shows a major impact on the systems' performance as they analyze different applications to different depths.

Figure 3.5, 3.6, 3.7, and 3.8 show 7-day snapshots of the TCP application-mixes in our four high-performance environments.[5] We see that the mixes are very distinctive. They are mainly affected by the availability and popularity of certain (internal as well as external) services. At MWN and UCB, HTTP is the dominant application for incoming and outgoing traffic, as it is in many networks providing access to a large number of users. Yet, at MWN the fraction of HTTP (58% of the total bytes) is considerably larger than at UCB (27%). A significant share of MWN's HTTP traffic is due to `dict.leo.org` which makes up 10-20% of all TCP connections. File-sharing protocols are relevant in both environments, which is typical for university networks. BitTorrent makes up a significant share at MWN and UCB; Gnutella does so only at UCB due to the corresponding port being blocked at MWN's perimeter. Yet, there are also considerable differences between the university environments. At MWN, RSYNC and FTP-DATA are among the top-5 outgoing applications; largely due to `ftp.leo.org`. At UCB, NNTP is a major service, incoming as well as outgoing; and CORAL (a peer-to-peer content distribution network; [FFM04]) is a major contributor in terms of outgoing traffic.

In contrast to the university environments, the traffic of NERSC and LBNL is dominated by large file-transfers using HSI, SSH and FTP-DATA. In these networks, there are also several TCP ports among the top-5 which do not correspond to any well-known application. As the time-intervals are relatively short in which these ports contribute a significant volume, they appear to be due to large individual connections (due to our measurement methodology, which considers source and destination ports, these may be HSI/FTP-DATA/SSH transfers).

Looking at how the traffic changes over time, we see strong time-of-day and time-of-week effects. Interestingly, the time-of-day effects are slightly shifted in time between the networks: at MWN, UCB, and LBNL/incoming the peaks of the total bytes tend to be in the early afternoon, while at LBNL/outgoing and NERSC the peaks are shortly after midnight. Most likely, this is due to application characteristics: in the former environments, interactive traffic dominates whereas the file transfers at LBNL/NERSC can be scheduled for off-hours.

There are applications whose fraction is rather stable over time (e.g., incoming NNTP at UCB and outgoing HSI at LBNL), and applications which are fluctuating yet do not show any obvious time-dependency for the major share of their volume (e.g., incoming SSH at MWN and outgoing RSYNC at MWN). Moreover, we see effects which appear

[5] The plots show GB per 10-minute intervals. These values are estimated based on TCP packets sampled at a rate of 1:4096 using a BPF-based sampling scheme. While not perfect, [Gon05] concludes that the scheme's accuracy is sufficient for such aggregations. We show the five top applications in terms of total bytes seen during the shown week, individually calculated for incoming and outgoing traffic per network. Applications are defined by TCP ports, considering both source and destination ports. The volume shown for FTP-DATA only includes the traffic on TCP port 20, not that on dynamically negotiated ports which also can make up a significant share. For HSI we joined the TCP ports 7500-7504. The numerical ports shown do not correspond to any particular well-known application. The time axes show the environments' local times. We note that the shown week is not necessarily representative.

to represent unusual situations; LBNL's outgoing HSI traffic and NERSC's incoming HSI traffic drop significantly on Thursday/Friday. Most probably, there is a single cause responsible for both of these as there is quite some traffic going from LBNL to NERSC usually.

Another observation is that a large fraction of the traffic is not classifiable by examining TCP ports. While the plots show only the top-five applications, each of the others contribute at most 4%-5% to its environment's total bytes, sometimes considerably less. Therefore, no other individual port can make up a significant amount of traffic, i.e., the unclassified bytes are distributed across a range of different TCP ports. A large fraction of them are certainly due to FTP-DATA transfers on dynamically negotiated ports. At UCB, there is also a considerable amount of peer-to-peer traffic on non-fixed ports.

For network intrusion detection, this observation is important to keep in mind as NIDSs typically trigger their protocol decoding based on a connection's destination port. With such a scheme, a large fraction of the traffic is only examined with the system's protocol-independent analysis (e.g., signature matching on raw packet payload).

For the performance of a NIDS, the application-mix is also crucial due to its impact on other traffic characteristics like connection duration and volume. In general, network traffic is "heavy-tailed": most network connections are quite short, with only a small number of large connections (the heavy tail) accounting for the bulk of the total volume [PF95]. Yet, the degree of "heavy-tailedness" varies. In [KPD+05] we show that 12% (NERSC, LBNL) to 15% (MWN) of all connections have a total size larger than 20 KB. Yet, the percentages of bytes per connection diverge significantly: if we cutoff all connections after the first 20 KB, we drop 87% of the bytes for MWN, 96% for LBNL, and even 99.6% for NERSC. This indicates that at MWN small connections account for a significantly larger fraction of the total traffic than at NERSC. At MWN, many of the small connections are due to `dict.leo.org`: 75% of its connections are smaller than 20 KB. At NERSC, the large file transfers make up almost all of the traffic's bytes.

Another characteristic of an environment is the ratio between incoming and outgoing traffic. This affects the performance of a NIDS if it analyzes outgoing traffic to a different depth than incoming. In general, in many networks there is more incoming traffic than outgoing. However, even a small set of popular services can change the picture. During the week shown in Figure 3.5, the ratio of incoming and outgoing bytes at MWN was roughly equal. However, when ignoring the systems of the `leo.org` domain (in which only about 15 hosts are non-negligible in terms of volume), there was twice as much incoming traffic than outgoing. At NERSC, we regularly see five times as much incoming traffic than outgoing. This is due to the super-computers crunching large volumes of input and returning condensed results.

There are also more subtle differences in environment characteristics. In [DP05], the authors observe that MWN shows a significantly higher degree of fine-grained packet reordering, compared to UCB and LBNL. For their work, this is important as they devise a TCP stream reassembly mechanism which is robust against attacks and suitable for hardware implementation. The increased reordering at MWN requires a 2-6 times larger buffer.

Given a particular environment, aggregated characterizations such as the application-mix tend to be relatively stable over medium-scale time periods (weeks) since usage patterns change only gradually. However, on smaller scales (hours to days) unusual situations can lead to noticeable changes. Any down-time of `dict.leo.org` or `ftp.leo.org` impacts MWN's traffic pattern significantly. Similarly, massively misbehaving software may be observable in the overall statistics (once we observed one of our local hosts generating 100s of thousands of connection requests; a user was testing a new peer-to-peer client). Furthermore, attack traffic can change the picture considerably: denial-of-service floods using spoofed source addresses can generate many thousands of new (apparent) flows per second [MVS01], greatly altering the total traffic pattern, as can worm propagation [MPS+03, SPW02].

Independent of such extreme situations, on even smaller time-scales (seconds to minutes) network traffic is very bursty. It exhibits an immense variability which essentially makes its precise characteristics unpredictable. The widespread prevalence of strong correlations and "heavy-tailed" data transfers [WTSW97, FGW98] regularly leads to sudden significant peaks of, e.g., traffic volume. It is important to observe that such peaks occur frequently; they do not represent anything unusual. Therefore, a NIDS needs to cope with them robustly rather than being built for the "average case".

Turning to traffic content, we also encounter diversity. Our main observation is that in a heterogeneous high-volume network stream, there is ample opportunity for any corner-case situation to manifest. Network protocol specifications define what constitutes a compliant conversation. Yet, inevitably they often leave room for interpretation. Sometimes RFCs contain "should"-clauses without defining how to act if the "should" does not hold. In other cases, people (and companies) introduce extensions to a protocol without following the standard RFC process. Even worse, we regularly encountered situations which, according to the relevant RFC, were plain wrong. Nevertheless, there are protocol implementations which generate such traffic, and apparently it works as otherwise they would not be in use. [Pax99] refers to such broken traffic as "crud". In the definition of the IP protocol [Pos81], a principle is formulated which, essentially due to such crud, became a generic design guide-line for network protocol implementations: *"In general, an implementation should be conservative in its sending behavior, and liberal in its receiving behavior."* Such defensive design is particularly important for a NIDS as in addition to all the "usual" crud, an attacker may also deliberately construct non-conforming traffic (see §2.4.3).

3.4 Summary

We presented the seven network environments which we use as our basis in the following chapters. We discussed their network infrastructure and summarized main characteristics such as internal/external bandwidth, and firewall policy. Four of the environments (MWN, UCB, LBNL, NERSC) match our criteria of a *high-performance* network: their upstream capacity is at least 1 Gbps, and their monthly external traffic is in the order of tens of terabytes.

We discussed the environment's policies and threats as well the immense diversity of their traffic. Our main observations are:

Site policies tend to be liberal. Most of our networks use rather liberal site policies, giving the users a large degree of freedom. Such policies tend to be informal and thus difficult to convert into device polices.

Liberal site policies are often supplemented with experience-based policies. Many networks supplement the liberal site policies with *experience-based* policies: rules specifying activity which, according to the network operator's experience, *tends* to be malicious. Experience-based policies are monitored with a combination of misuse and anomaly detection approaches, often concentrating on high-volume activity.

Internet-scale undirected attacks are a major threat. Any device connected to the Internet is under constant attack by Internet-scale scans. Due to their automated nature, these attacks pose a significant threat.

Worms and resource-misuse are most easily detected by their consequences. Hosts change their traffic pattern when they are infected by a worm or compromised to misuse their resources. Such a change is often more reliable to detect than the initial exploit.

Active response can reduce the number of incidents. Many administrators are reluctant to deploy automatic response. Yet, sites' experience shows that blocking scanners is of great benefit. Accordingly, in-line systems have indeed a significant potential to prevent attacks.

Traffic characteristics differ significantly between environments. The traffic pattern of an environment depends on the availability and popularity of (internal and external) services. Therefore, they differ significantly between networks, impacting the performance of a NIDS.

High-volume traffic is unpredictable. A high-volume traffic stream presents ample opportunity for traffic variations to manifest. Thus, a NIDS needs to cope with unexpected situations.

In the near future, most of the high-performance environments that we have examined are going to increase their upstream capacity to 10 Gbps. At such rates, commodity hardware is going to become a real bottle-neck. Therefore, the deployment of customized

monitoring hardware, like network processors or FPGAs, for performance-critical tasks will become crucial. This is an important area for future research.

Another interesting research direction is understanding the similarities and differences between different classes of environments. For example, all of the networks discussed in this chapter are part of research environments. Enterprise environments tend to have less liberal site policies and thus differ. Characterizing the differences between research and commercial networks has not yet received much attention.

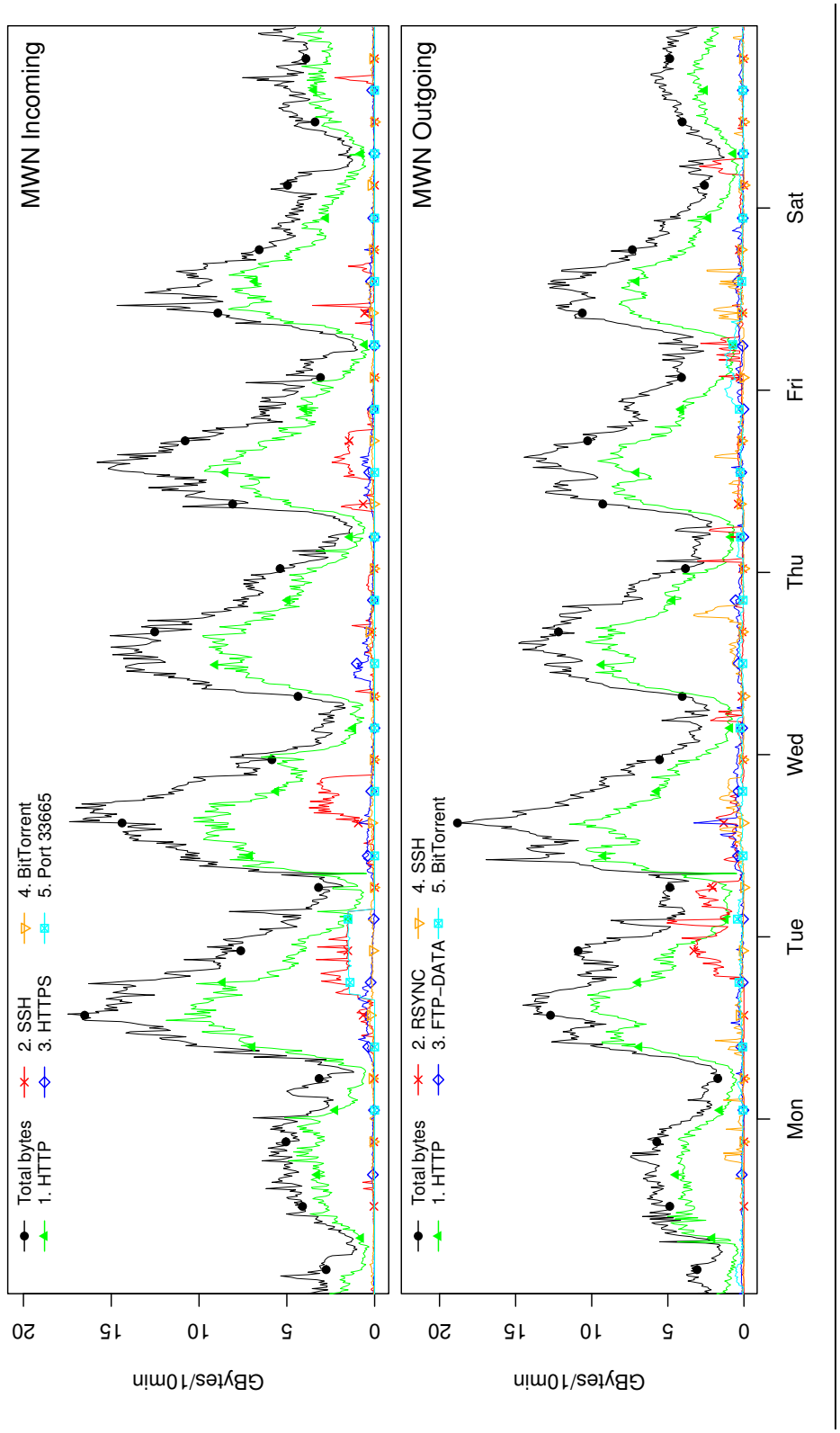


Figure 3.5 Snapshot of TCP application-mix at MWN (May 8 2005 - May 14 2005).

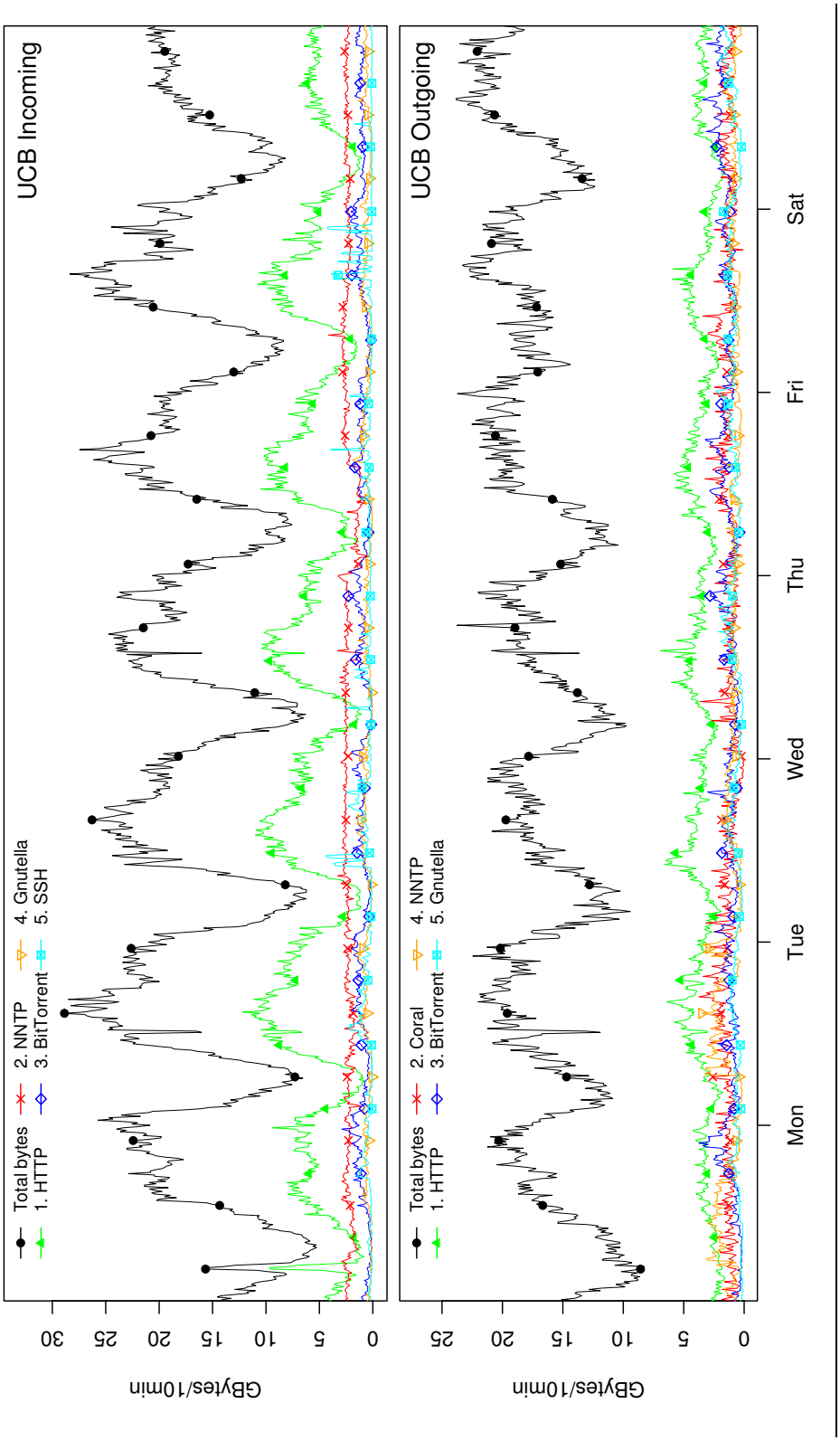


Figure 3.6 Snapshot of TCP application-mix at UCB (May 8 2005 - May 14 2005).

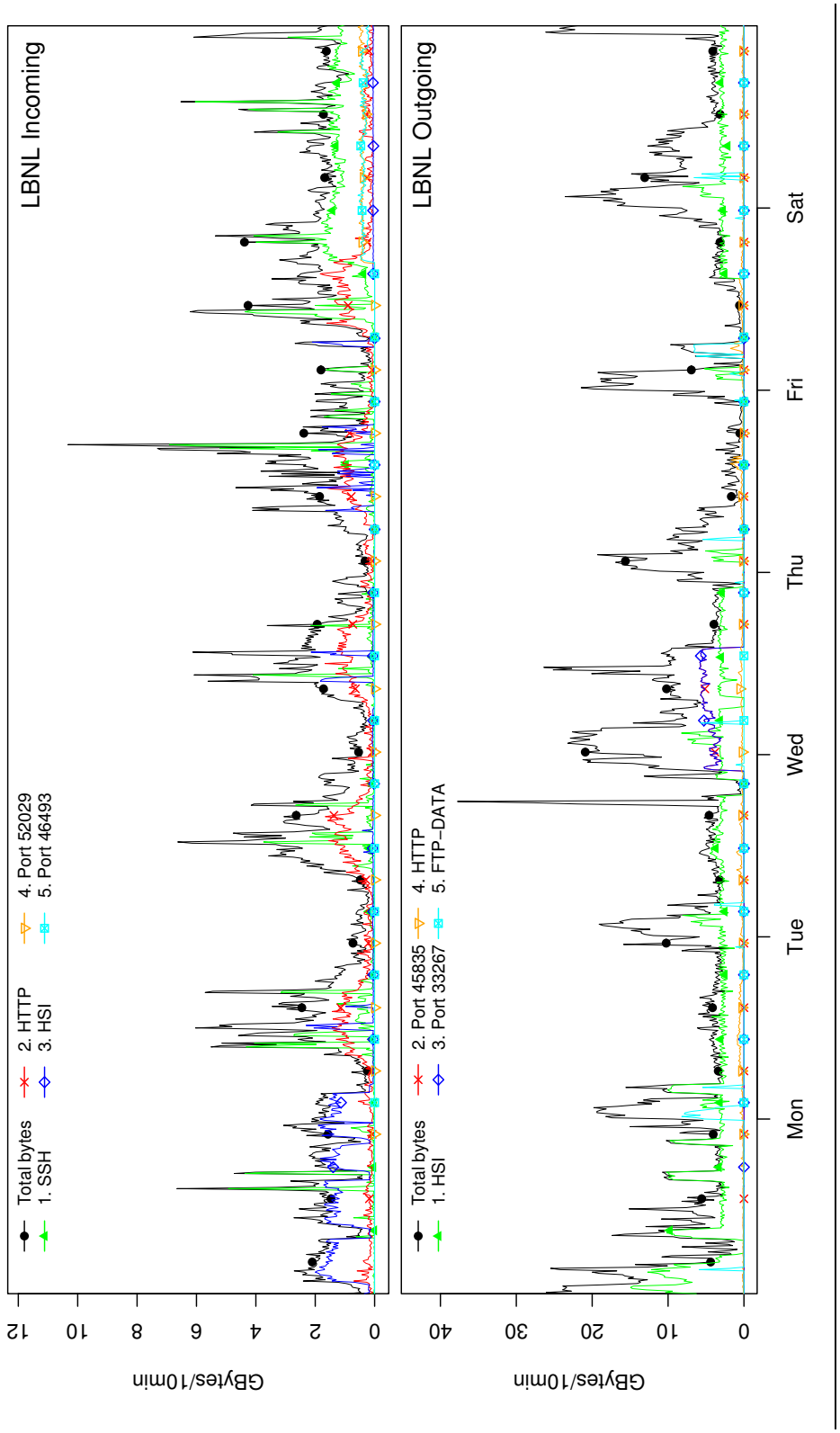


Figure 3.7 Snapshot of TCP application-mix at LBNL (May 8 2005 - May 14 2005).

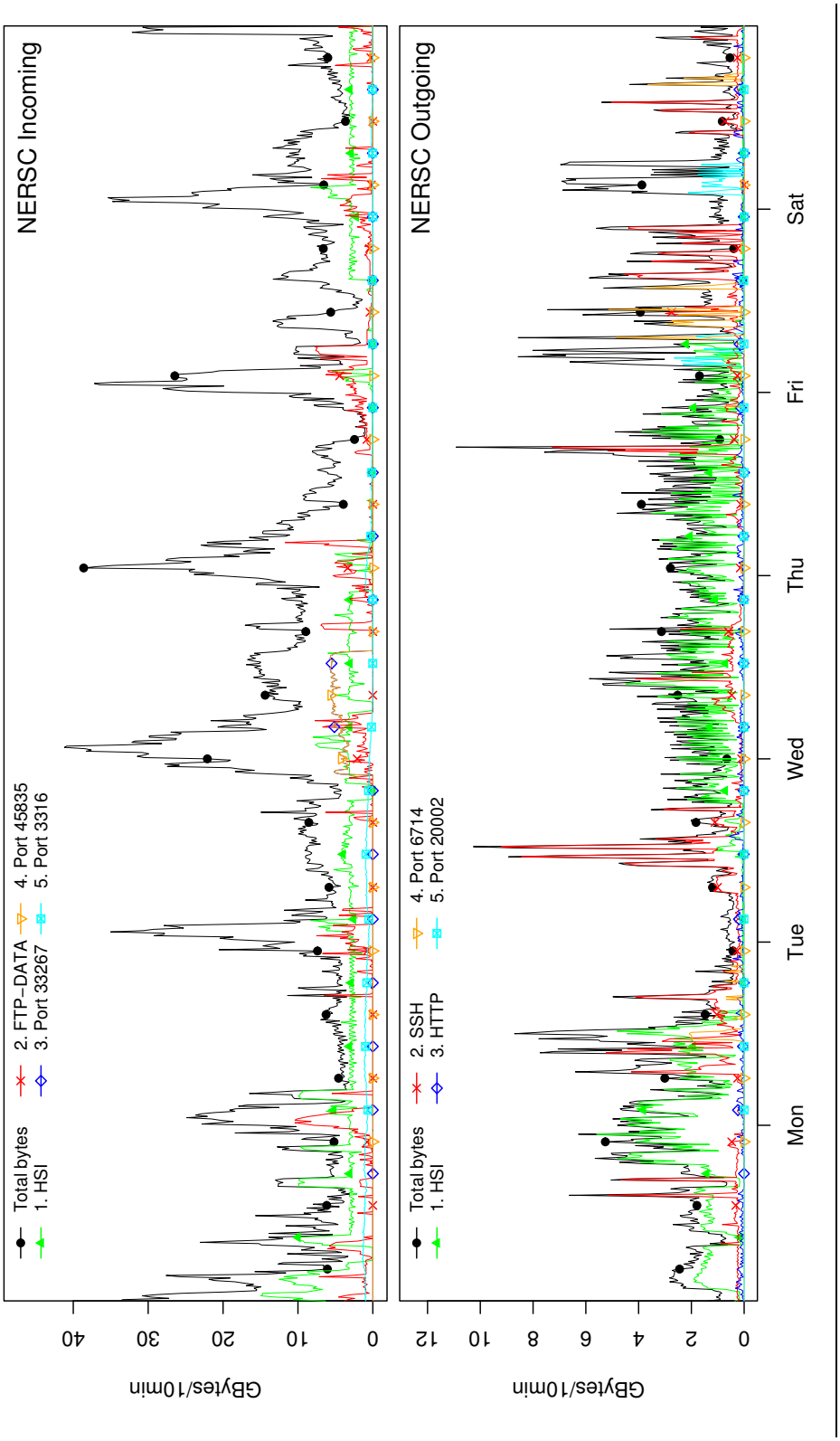


Figure 3.8 Snapshot of TCP application-mix at NERSC (May 8 2005 - May 14 2005).

4 Operational Deployment

> There comes a time when for every addition of knowledge you forget something that you knew before. It is of the highest importance, therefore, not to have useless facts elbowing out the useful ones.
>
> — *Sir Arthur Conan Doyle, "A Study in Scarlet"*

In large-scale environments, network intrusion detection systems face extreme challenges with respect to traffic volume, traffic diversity, and resource management. While crucial for acceptance and operational deployment, the research literature often ignores the practical difficulties that arise in such networks. In this chapter, we first report our practical experiences with deploying NIDSs in the environments discussed in §3. Then we systematically study the resource usage of one of the systems. Finding that it lacks necessary mechanisms to cope with high-volume traffic, we devise several improvements which enable us to operate the NIDS successfully in our environments.

4.1 Overview

When deploying NIDSs in high-performance environments, one is faced with practical difficulties. If in such a network one simply installs and runs an untuned NIDS, almost surely it will be unable to effectively cope with the amount of traffic. Some systems immediately consume the entire CPU, leading to excessive packets losses; others quickly exhaust all available memory. Signature-based systems tend to overwhelm one with alerts. Such practical difficulties rarely see investigation in the research literature: NIDS vendors often have a commercial interest in downplaying the difficulties and keeping private their techniques for addressing them, and researchers seldom have opportunities to evaluate systems in operational high-performance environments.

We offer such a study. We start with a report of our experiences from working with four systems: Snort, Bro, Dragon, and IntruShield. We discuss their alerting/logging capabilities, signature quality, user interfaces, resource demands, evaluation difficulties, artifacts introduced by the monitoring environments, and, finally, expose the problems that even small programming deficiencies can cause.

Next, we concentrate on the resource demands of one specific NIDS. In the context of the Bro system, we systematically identify and explore key factors with respect to memory management and efficient packet processing. We find that three factors dominate overall resource consumption: *(i)* the total amount of state kept by the system, *(ii)* the

traffic volume, and *(iii)* the widely varying per-packet processing time. While these factors certainly are not surprising by themselves, the key is understanding them for tuning a NIDS and adapting it appropriately to the environment; there is a somewhat atypical trade-off between CPU/memory and detection rate that is rather illuminating. On the one hand, our insights help us gauge the trade-offs of tuning a NIDS. On the other hand, they motivate us to explore several novel ways of reducing resource requirements. These enable us to improve NIDS state management considerably as well as to balance the processing load dynamically.

Overall this work provides us with a NIDS much more suitable for use in high-volume networks, both in terms of raw capabilities and greater ease of tuning. In addition, our study illuminates complexities inherent in analyzing, tuning and extending systems that must process tens-to-hundreds of thousands of packets per second in real-time.

In §4.2 we report our experiences with the deployment of different NIDSs. In §4.3 we present our methodology for measuring the memory and CPU usage of the Bro NIDS. Based on a set of traces, we then analyze Bro's resource usage in §4.4. Finally, in §4.5, we present several enhancements to Bro which together enable us to now operate Bro successfully in our high-volume environments.

4.2 Experiences with Network Intrusion Detection Systems

Typically, the deployment of a NIDS is an iterative process. When installing a system, it needs to be customized to the local site policy. However, beforehand it is difficult to anticipate operational results such as number of generated alerts and resource demands. Therefore, initially the system is usually equipped with a default configuration. Then, during the next couple of weeks, the parameters of the NIDS are tuned as required, e.g., inappropriate signatures are disabled and time-outs for the state management adjusted. Eventually, a stable configuration should be reached which provides the operator with a tractable number of alerts without unnecessarily missing attacks. Inevitably, the configuration will require new modifications over time to accommodate changes in, e.g., traffic patterns and signatures.

In this section, we report the observations we made while following this process with different NIDSs. In the MWN environment, we deployed four systems: Bro, Snort, Dragon, and IntruShield (see §2.5). Bro was our primary object of research and we used—and improved—it throughout our work. We ran Snort for a total of a couple of weeks; the LRZ is also deploying Snort. Dragon and IntruShield were installed at MWN for evaluation purposes, roughly two months each. In the other environments discussed in §3, Bro is the primary network monitor. At UCB, LBNL and NERSC, Snort supplements Bro's detection. UCB also uses an IntruShield system. During the Dragon evaluation period, we also installed a Dragon sensor at WGN.

We note that it is not our intent to compare the NIDSs in terms of quality or efficiency. As discussed in §2.4.5, evaluating NIDSs is non-trivial, and such a study is not the focus of this work. Yet, in §4.2.5, we discuss some of our *problems* with comparing aspects of the NIDSs. In later chapters, when we evaluate our improvements to network intrusion detection, we need to keep these difficulties in mind.

4.2.1 Logs, Alerts and Forensics

The primary output of a NIDS is a list of alerts. When we installed the three primarily signature-based NIDSs—Snort, Dragon, and IntruShield—at MWN, we were always immediately flooded with alerts (tens of thousands a day). By generating such a volume of alerts, the Dragon system actually filled up its 32 GB hard disk every 3-4 days and then ceased to work until the disk was cleaned up manually.[1] On the other hand, the Bro system, which did not deploy a signature library, remained relatively quite, often reporting mainly scans. Oddly enough, this also turned out to be unsatisfactory: in such a large research network, there certainly has to be *something* going on.

Hence, all systems needed tuning; a fact generally acknowledged by users and vendors of NIDSs alike. For the signature-based systems, tuning primarily means figuring out which signatures are irrelevant in the system's environment and disabling them. To ease the process, the systems group signatures into different categories which may be disabled as a whole. In general, such signature tuning is an iterative process, often taking several weeks until the number of alerts reaches a manageable volume. We observed that the disabling of a signature is mainly due to one of two reasons: *(i)* we were sure that a signature is not relevant for us; *(ii)* we were uncertain whether it applied but the signature triggered too many alerts. From a security point of view, the latter is a rather awkward reason. However, often signatures lack the tightness required to reliably separate attacks from benign traffic. We discuss signature quality in more detail in §4.2.2. Moreover, many alerts reported attacks which were *unsuccessful*. Numerous attackers are automatically scanning the Internet for vulnerable hosts, trying exploits on every potential victim by brute-force. Most administrators have stopped bothering with such probes as long as there is no success. Yet, NIDSs are mostly unable to differentiate between successful and unsuccessful attacks. We tackle such problems in §6.

To tune Bro, we need a considerably deeper understanding of the site policy to effectively configure the system. The system ships with a large set of default scripts which need to be adapted. This requires very specific information, e.g., Bro's connection analyzer checks which inbound/outbound services are permitted, and the login analyzer reports user commands which are considered malicious. For a large heterogeneous network like MWN, it is impossible to provide a NIDS with such a level of detail. Thus, the typical approach is to pick out hosts for which we do have the required knowledge to reproduce their policy. For the rest of the network, we need to resort to more general checks like scan detection. Note that this scheme does in fact match with the liberal policies of the backbone networks (see §3.3.1).

A major advantage of Bro compared to the other three systems is its scripting capability which provides unmatched flexibility. With Snort, Dragon and IntruShield we are restricted to the built-in detection capabilities. With Bro it is fairly easy to implement arbitrary new schemes. Once, for instance, when a new worm broke out, we devised a script which detected infected hosts based on their scanning pattern. Moreover, Bro includes a very generic alert filter. Before an alert is escalated, it first passes through a

[1] Closer inspection of Dragon's hard disk eventually revealed a Perl script which expires old logs and may be inserted into the system's crontab manually. However, apparently it did not support the logging scheme of the current version.

customizable list of script-level filters which may reduce its priority. At MWN, we use this mechanism to, e.g., separate locally executed scans from remote scans.

In terms of logging, Bro is unique in that it logs different kinds of *activity* rather than just alerts. Such logs include one-line summaries of every connection it has seen, accesses to HTTP and FTP servers, and DNS queries. The level of detail is configurable. A couple of times, these logs proved to be invaluable for incident analysis; often in cases in which, during the attack, the NIDS had not raised an alert. In particular, the connection summaries provide a great deal of information. For instance, once at USB several machines were compromised by exploiting a bug in SSH [CVE02b]. While the network operators were able to identify some of the victims by observing them participating in DDoS attacks, it was hard to tell if there were others just waiting to be activated. Knowing some of the victims, we used Bro's connection summaries to identify the SSH sessions which carried the initial exploits. We then checked for connections to other internal hosts with similar characteristics, and, in fact, found several additional ones. All of them proved to be compromised. Apart from security monitoring, such activity logs are also well-suited for more general kinds of traffic analysis; in [SF02] we leverage them to assess the quality of Cisco's NetFlow [Cis] data.

Given an alert, it is often difficult to understand what exactly has been the trigger. It requires a great deal of time and expertise to assess whether a policy violation had indeed occurred. When comprehending the cause of an alert, the NIDS's user interface plays a major role; accordingly we discuss user interfaces in more detail in §4.2.3. In general, we observe that with more data it is easier to track the incident down. To this end, one of the most helpful resources are packet traces. While the logs of NIDSs condense the network activity, packet traces provide us with the full information. Unfortunately, due to the high volume in medium- and large-scale environments, capturing full packet traces quickly gets impractical. Thus, the NIDSs provide ways to capture only a subset of the traffic. All four examined NIDSs can record just the packets which triggered alerts. While better than nothing, a single packet often does not provide enough context to understand an incident. For connections that triggered an alert, Dragon may also store the payload stream following the match up to a configurable amount of bytes. This is much more helpful, yet it still depends on an alert being raised. Bro provides the ability to capture all packets which it has inspected. However, this may include a large part of the traffic, requiring significant storage. Yet, due to the internal packet filter, often relevant content is still missing. In practice, we often found the need to inspect connections which were not recorded (and, in particular, had not set off an alert themselves), e.g., a host may have been scanning our network and we would have liked to inspect the traffic of all the connections it has successfully established. To provide such an after-the-fact view of network activity, some sites in fact routinely bulk-record their full traffic, archiving it for a short time period in case it turns out to need closer inspection. At LBNL, for instance, there is operational bulk-recording in place. Yet, to be feasible all Web traffic (and more) is excluded due to its volume. To address such problems, we have developed a smarter bulk-recorder that is able to buffer multiple days of "interesting" traffic even in large-scale networks [KPD+05].

4.2.2 Signature Quality

For a signature-based NIDS, the quality of its signatures is a major factor. Due to the widespread prevalence of Snort, its signatures comprise the most comprehensive signature set that is openly available. Many people contribute to it, and new signatures are frequently published.

However, via personal communication we heard from several operators of large networks that Snort's default signatures lack the tightness required for an effective operational deployment. In fact, a common way of using Snort is to deploy only a very small set of *custom* high-quality signatures, completely skipping the shipped set. Often, such an operational set includes only a handful of highly specific entries.

We examined Snort's open-source signatures, and indeed they show a lack of quality. Many of its signatures are too general. For example, Snort's signature #1560:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
  (msg:"WEB-MISC /doc/ access"; uricontent:"/doc/";
  flow:to_server,established; nocase; sid:1560; [...])
```

searches for the string `/doc/` within URLs of HTTP requests. While this signature is indeed associated with a particular vulnerability (`CVE-1999-0678` [CVE]), it only makes sense to use it if you have detailed knowledge about your site (for example, that there is no valid document whose path contains the string `/doc/`). Otherwise, the probability of a signature match reflecting a false alarm is much higher than that it indicates an attacker exploiting an old vulnerability. Similarly, the Snort signature #1042 to detect an exploit of `CVE-2000-0778` [CVE] searches for "`Translate: F`" in Web requests; but it turns out that this header is regularly used by certain applications. Another problem with Snort's default set is the presence of overlapping signatures for the same exploit. For example, signatures #1536, #1537, #1455, and #1456 (the latter is disabled by default) all search for `CVE-2000-0432`, but their patterns differ in the level of detail. In addition, the vulnerability IDs given in Snort's signatures are not always correct. For example, signature #884 references `CVE-1999-0172` and Bugtraq [Bug] ID #1187. But the latter corresponds to `CVE-2000-0411`.

We believe that the major reason for Snort's low signature quality is its very liberal community model: many people contribute signatures to the set which work *for them*. Yet, apparently, there is a lack of central quality assurance to exclude signatures not suitable for general deployment. In fact, Snort's developers seem to have arrived at similar conclusions. Recently, the signature set has been split into *Sourcefire VRT Certified Rules* and *Community Rules*, with the former containing signatures verified and approved by Sourcefire's *Vulnerability Research Team*.[2] Moreover, Sourcefire and *Bleeding Snort* [Bld], an independent community effort to devise signatures, have joined

[2] For customer's of Sourcefire, such certified signatures have already been available for quite some time, and they still get the certified set up to five days in advance. Certified signatures are not covered by the GPL (as Snort itself) but Sourcefire's proprietary *VRT Certified Rules License Agreement* which, essentially, forbids commercial redistribution.

forces by building the *Open Source Rules Consortium*. However, we note that the current *VRT Certified Rules* still contain all of the examples mentioned above. It appears that existing signatures have not been checked thoroughly. In fact, we believe it would be best to remove many of the very old signatures that match on vulnerabilities which are unlikely to be still relevant.

Turning to Dragon and IntruShield, their signature quality is higher in the sense that their patterns are not as general as, e.g., Snort's `/doc/`. Unfortunately, IntruShield's signatures are not openly available, thus we could not inspect them. Dragon's signatures are open, and while we did not examine them systematically, they appear to be sound (as long as we ignore the general limitations of pure byte-level pattern matching; see §6).

A large fraction of both systems' signatures detect the use of certain services (e.g., IRC or file-sharing) rather than attacks. Naturally, it depends on the site policy if such signatures should be enabled. Moreover, these signatures are often easy to mislead. With IntruShield, we were able to trigger false positives by issuing HTTP requests containing a certain peer-to-peer client as `User-Agent`.

Furthermore, with both commercial systems we found problems with the descriptions of the signatures. Such descriptions provide the user with a signature's objective as well as typical problems. With IntruShield, we found, e.g, two different signatures which were linked to the same plain text description. Inevitably, it was wrong for one of them. With Dragon, several alerts per minute were triggered at MWN by a signature for which "there should not be any false positive". Closer inspection of the signature showed that it was supposed to match malformed DNS packets issued by a particular DDoS-tool. However, even in the small WGN network such packets are regularly generated by a Solaris system.

Similar to the other systems, Dragon categorizes its signatures into groups. However, it uses two different schemes at the same time: first, it provides a couple of very broad categories, e.g., "Misuse" and "Attacks". Second, the naming scheme of the signatures (e.g., "IRC:JOIN-ALL-PORT") implicitly introduces a more fine-grained grouping via the use of common prefixes. However, while the coarse-grained groups can be excluded as a whole, it is not possible to use these prefixes for generally disabling, e.g., all IRC signatures—which would be much more useful than disabling all "Misuse" signatures.[3]

4.2.3 User Interfaces

There are two main types of user interfaces: *(i)* ASCII-based configuration and log files, accessible with command line tools, and *(ii)* interactive graphical user interfaces. Snort and Bro are purely-ASCII based. IntruShield comes with a Java-based GUI and does not provide any low-level interface.[4] Dragon combines both approaches: while its GUI appears to be the main interface, one can log into the system with SSH and inspect its

[3] Enterasys confirmed this but mentioned that a future version may include such a feature. We got similar responses a couple of times while working with the IntruShield and Dragon systems.

[4] You can externally access its MySQL database though.

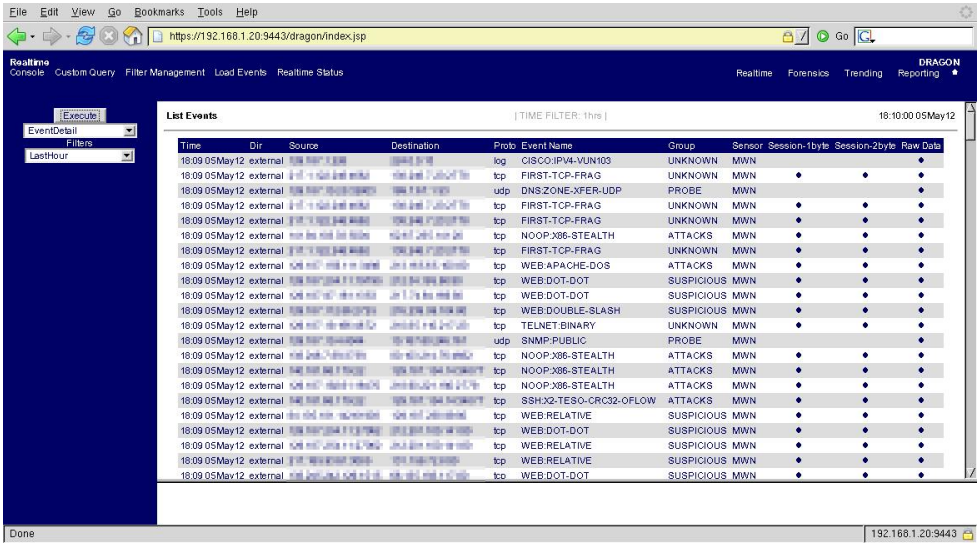
ASCII-logs and XML-configurations. Moreover, a set of command line tools is installed on the system which access Dragon-internal state.

The preference for a particular kind of user interface is, first of all, a matter of taste. If one is used to analyzing data with standard Unix tools such as *grep*, *sed*, and *awk*, with a GUI one will very likely miss the accustomed flexibility; typically it allows only a fixed predefined set of analyses. On the other hand, being forced to rummage through ASCII-logs manually will be rather frustrating if one is not familiar with Unix command line tools. That said, we believe that apart from personal preference, there are advantages as well as major short-comings of each of the interfaces provided by the four systems, and we discuss them in the following.

Snort and Bro feature similar interfaces. Their configuration is provided as ASCII files, editable with any standard editor. They log their results into ASCII-files as well. The the major difference between the systems is that Bro logs significantly more information (see §4.2.1 for a description of Bro's activity logs). Internally, Dragon works similar to Snort and Bro. Its configurations files are written in XML, and it also stores its alerts in ASCII-files. For an experienced Unix user, it is easy to filter and correlate such ASCII log files in a multitude of different ways. However, the user needs to devise such analyses himself, potentially putting them into scripts if they are needed on a regular basis. While Bro ships with a few of such scripts, most users come up with their own to fully utilize the flexibility. For Snort and Bro, this also holds for reporting and maintenance tasks, including mailing-out alerts and archiving log files. For less technically-skilled users, setting this up may be too much of a hassle. Yet, even for advanced users there is a major drawback of command-line-based data analysis: the lack of interactivity. When analyzing NIDS logs, we regularly encountered situations which required more context to comprehend. With a command-line interface, we then needed to come up with another chain of commands whose execution may take tens of seconds to minutes. Moreover, if our guess about where to find the necessary context turned out to be wrong, we needed to repeat the step. On the other hand, a GUI could theoretically provide us with ability to quickly jump back and forth between different views and levels of details. It could supplement the log data with external information and pre-calculate correlations for quick access.

Unfortunately, the two commercial systems that we examined do not take advantage of this potential currently. Both, IntruShield and Dragon, provide browser-accessible Java-based GUIs for data analysis; Figure 4.1 shows a snap-shot of the Dragon tool reporting details about recent alerts. The responsiveness of both GUIs is rather low, in particular when processing large volumes of log information. In both cases, this may be due to the choice of Java as underlying technology. However, delays significantly limit the experience of interactivity. If each step may take a couple of seconds to complete, one quickly starts to restrict one's actions to the minimum possible. Yet, ideally a GUI should provide a playful way of exploring data even in directions that do not look promising at first glance. In terms of functionality, both IntruShield and Dragon do not provide much more than a standard database-browser. Both systems keep their information (also) in databases, and their GUIs are essentially restricted to selecting,

Figure 4.1 Snapshot of Dragon's graphical user interface for alert analysis.

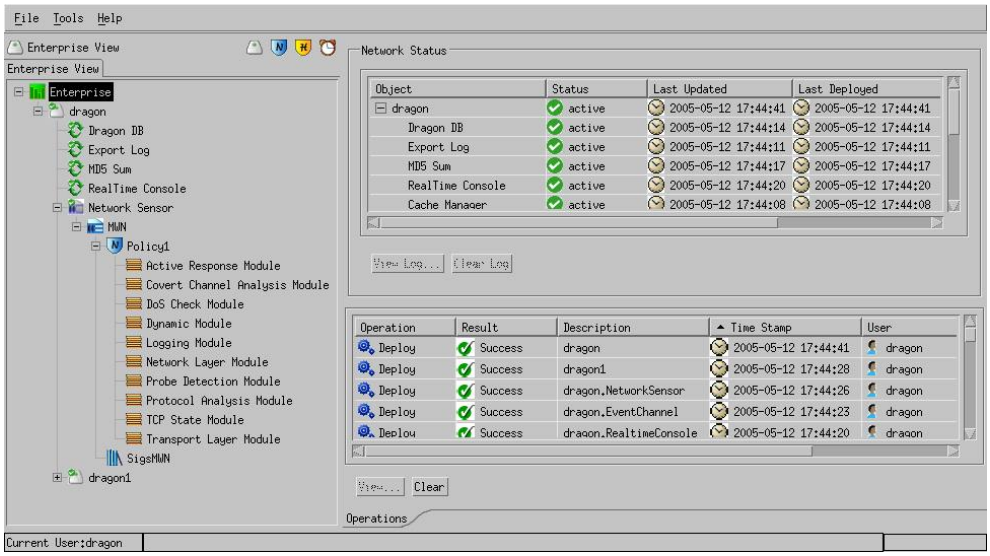


aggregating and displaying alerts based on different criteria. They do not appear to exploit many specifics of the intrusion detection domain. In particular, we have not yet seen any sophisticated correlation techniques. While both systems are able to produce longer-term statistics and attack trends, the practical use of such reports is doubtful.

With the Dragon system, there are *separate* interfaces for configuring the system and analyzing the logs. While the latter runs in a Web browser, the former is a stand-alone Java application. From a usability point of view, this is awkward. For example, when tuning the system, we first use the browser to identify a signature which produces unintended alerts. Inside the browser, we may also get a short informal description of the signature's intent. Next, we are required to switch to the stand-alone client and find the corresponding signature there. Then we can look at the signature's definition to understand precisely when it matches. Finally, we can either fully deactivate the signature or change back to the browser and, manually, create a custom filter which will hide the alerts in the future. If we choose to fully disable the signature, we need to send the new configuration to the sensors. This whole process is tedious, and the lack of responsiveness in both GUIs (analysis and configuration) does not ease the task. We believe that getting to see the signature and if necessary disabling it, should take not more than two quick mouse clicks.

The Dragon GUI also exhibits another problem otherwise more usually seen with 3rd-party add-ons (e.g., for Snort): it lacks a tight integration with the detection components. Apparently, the GUI has been developed independently. The stand-alone configuration client acts like a remote control: it issues commands to the NIDS via a TCP connection, and then waits for replies. The problem is that this concept is visible to the user: the GUI contains a prominently placed box which shows the current state of sent and acknowledged configuration update commands (see Figure 4.2, bottom-right). We believe

Figure 4.2 Snapshot of Dragon's graphical user interface for configuration.



that for a security analyst such information is irrelevant. Apart from usability issues, there are typical technical problems with the remote-control approach: we have seen desynchronizations between NIDS and client, ignored updates, and problems which could only be fixed directly on the NIDS, e.g., by manually restarting it rather than having the client issuing a restart command.

While a good GUI has the potential to provide significant benefit, there are tasks which do not fit well into a graphical concept. We encountered an example in IntruShield: writing custom signatures. IntruShield's signatures can combine conditions by means of logical operators. In our opinion, opening an editor would be the most straight-forward way to provide the user with the ability to create such clauses. Instead, IntruShield offers the user a set of combo-boxes containing all possible conditions and operators. Thus, the user devises complex logical clauses by "clicking"—which is cumbersome.

A technical problem of IntruShield's browser-based GUI used to be that it worked only with Microsoft's Internet Explorer (and in fact only with a particular version of it). We have been told that newer releases are now also usable with alternative browsers such as Firefox, yet still only on Windows platforms. For some environments (e.g., MWN) such a limitation is a show-stopper.

4.2.4 Resource Demands

All NIDSs face a fundamental trade-off between resource requirements and detection rate: *(i)* in-depth analysis requires CPU time and *(ii)* comprehensive state needs memory. In both cases, the constraint of real-time processing limits the set of available techniques: neither detection nor state management may introduce significant delays. Otherwise, we would lose packets as only a small amount of them can be buffered.

In terms of processing performance, we observed that, in general, commodity PC hardware quickly reaches its limits even in medium-scale environments. At MWN, during the busy-hours both Dragon and Snort often missed more than 50% of the traffic when using their default configurations (as reported by themselves). While with Bro there is hardly any "default configuration", it is also not able perform more sophisticated kinds of analysis without incurring significant packet losses (see §4.4). With respect to the Dragon system, this is actually one of our most disappointing observations: Dragon ships as a custom-made box specifically build for use in high-performance networks. Yet, it still does not perform better in terms of efficiency than the open-source systems running on widely-available hardware. There are some options to reduce CPU requirements though which indeed mitigate the drops (like reducing the amount of reassembled TCP streams). We discuss such trade-offs in §4.4.6. Compared to Dragon, the IntruShield system, using its custom hardware, looks much more promising. While the system does not provide statistics about the amount of dropped packets, we are willing to believe that, in principle, such hardware is able to cope with the traffic in high-performance networks.

With respect to memory management, Snort, Dragon, and IntruShield all use fixed upper limits for the amount of stored state. On the other hand, Bro dynamically allocates memory as required—which, while more flexible, can lead to crashes due to memory exhaustion. Moreover, originally Bro chose to not expire a large share of its state ever [Pax99]. Instead, it relied upon checkpointing (see §2.4.4) which is a rather drastic way of flushing state. In §4.5 and §5.4.2, we show our approaches to avoid unconditional checkpointing. But first, in §4.4, we study the resource demands of Bro in more detail to understand where its bottlenecks are.

4.2.5 Evaluation Difficulties

When comparing two NIDSs, differing internal semantics can present a major problem. Even if both systems perform basically the same task—capturing network packets, rebuilding payload, decoding protocols—that task is sufficiently complex that it is almost inevitable that the systems do it somewhat differently. When coupled with the need to evaluate a NIDS over the same *large* input traffic (millions of packets), which presents ample opportunity for the differing semantics to manifest, the result is that understanding the significance of the disagreement between the two systems can entail significant manual effort.

One example is the particular way in which TCP streams are reassembled. Due to state-holding time-outs (see §2.4.4), ambiguities (see §2.4.3) and non-analyzed packets (which can be caused by packet drops, or by internal sanity checks), TCP stream analyzers will generally wind up with slightly differing answers for corner cases.

Snort, for example, uses a preprocessor that collects a number of packets belonging to the same session until certain thresholds are reached and then combines them into "virtual" packets which are passed on. The rest of Snort is not aware of the reassembling and still sees only packets. Bro, on the other hand, has an intrinsic notion of a data stream. It collects as much payload as needed to correctly reconstruct the next in-

sequence chunk of a stream and passes these data chunks on as soon as it is able to. The analyzers are aware of the fact that they get their data chunk-wise, and track their state across chunks. They are not aware of the underlying packetization that lead to those chunks.

Another example of differing semantics comes from the behavior of the application-layer protocol analyzers. Even when two NIDS both decode the same application-layer protocol, they tend to differ in the level-of-detail and their interpretation of corner cases and violations of the protocol (which are in fact seen in non-attack traffic; see §3.3.3). For example, both Bro and Snort extract URLs from HTTP sessions, but they differ in their interpretation in some situations as character encodings within URLs are sometimes decoded differently. The anti-IDS tool Whisker [Whi] can actively exploit these kinds of deficiencies (see §5.7.4). Similarly, Bro decodes pipelined HTTP sessions; older versions of Snort did not, they only processed the first URL in a series of pipelined HTTP requests.

Usually, the details of a NIDS can be controlled by a number of options. But frequently, an option in one system has no equivalent in the other, and vice versa. For example, the amount of memory used by Snort's TCP reassembler can be limited to some fixed value. If this limit is reached, old data is expired aggressively. Bro relies solely on time-outs. Options like these often involve time-memory trade-offs. The more memory we have, the more we can spend for Snort's reassembler, and the larger we can make Bro's time-outs. But how to choose the values, so that both will utilize the same amount of memory? And even if one does, how to arrange that both expire the same old data? Such hooks do not exist.

The result of these differences is varying views of the same network data. If one NIDS reports an alert while the other does not, it may take a surprisingly large amount of effort to tell which one of them is indeed correct. More fundamentally, this depends on the definition of "correct," as generally both are correct within their semantics. From a user's point of the view, this leads to different alerts even when both systems seem to use similar signatures. From an evaluator's point of view, we have to *(i)* grit our teeth and be ready to spend substantial effort in tracking down the root cause when validating the output of one tool versus another, and *(ii)* be very careful in how we frame our assessment of the differences, because this is to some degree a fundamental problem of "comparing apples and oranges".

The same applies for measuring performance in terms of efficiency. If two systems do different things, it is hard to compare them fairly. Again, the HTTP analyzers of Snort and Bro illustrate this well. While Snort only extracts URLs, Bro decodes the full HTTP session, including tracking multiple requests and replies (which entails processing the numerous ways in which HTTP delimits data entities, including "multipart MIME" and "chunking"). Similarly, Bro provides much more information at various other points than the corresponding parts of Snort.

But there are still more factors that influence performance. One of our main observations along these lines is that the performance of a NIDS can depend heavily on the particular input and even on the underlying hardware. In §6.3.2 we show several measurements which demonstrate such effects.

Aligning and comparing the output of two NIDSs is already difficult with open-source systems. It becomes even harder with closed-source commercial systems. These have to be understood by *observing* their output; one cannot read any authoritative source code. In particular, while they usually provide plenty of configuration parameters, often one cannot predict the consequences of choosing particular values. For example, both the Dragon and the IntruShield provide an option to specify the maximum number of simultaneous sessions for their state-holding engine. To comprehend the semantics of this parameter, at the very least we need to know *(i)* what precisely constitutes a session, and *(ii)* what happens if the session table gets full. Unfortunately, companies usually do not provide us with such details, and often even the companies' field technicians can only guess. About IntruShield, one such technician told us that, when the session table has filled up, the system would stop to accept new sessions. For the in-line IntruShield, this would essentially mean that it suspends connectivity. We prefer to not believe the technician's statement.

4.2.6 Artifacts of the Monitoring Environment

So far, we have discussed aspects of the NIDSs themselves. However, we find that high-volume environments also stress the monitor environment, i.e., the monitoring router, the NIDS's network interface cards, and the OS-level packet handling.

First of all, there are some general capacity limitations. At UCB, the traffic of several routers is simultaneously monitored by merging their Gbps streams using an RSPAN-VLAN [RSP]. This can exceed the monitor's Gbps capacity. Indeed, we see both missing and duplicated packets in the NIDS's input stream. We believe that this is due to the RSPAN setup. While only a single router is monitored at MWN, both directions of the network's Gbps upstream link are merged into one unidirectional monitor link using a SPAN port. While the available capacity is usually sufficient, the router does report occasional buffer-overruns (i.e., causing the monitor to miss packets). To overcome these limitations we intend to switch from a SPAN port to optical taps. Yet this introduces the problem of merging two traffic streams into one within the NIDS's system. This requires tight synchronizing between the two streams to maintain causality (e.g., SYN-ACKs must be processed after the corresponding SYN). Endace's DAG cards (see §4.5.4) support such a synchronization.

In the past, an older MWN-router exhibited another strange behavior: it randomly duplicated a fraction of the packets. The only differences between the two exemplars were a decreased hop-count and different link-layer addresses. This suggests that the router put the affected packets on the monitoring port at two different times: once when they arrived on an input port and another time when they departed on an output port. A support-call did not produce any explanation for this behavior.

The default Gbps NICs in the MWN monitoring systems are Intel Pro/1000 MF-LX. To avoid packet losses, on the FreeBSD machines we patched the kernel to increase the NIC driver's internal receive buffers. Moreover, we patched the packet-capture subsystem to increase its buffers by three orders of magnitude. On our Linux systems, we noticed an inferior packet-capture performance, often seeing significant drops when a

similar FreeBSD system was running fine. We temporarily switched to two other Linux packet-capture implementations [Der04, MMP]. Yet, both alternatives did not work reliably: with [Der04], with certain filter expressions the packet filter discarded packets which should have been preserved; with [MMP], packet timestamps got corrupted when using libpcap's non-blocking mode. While we are still investigating capture performance on Linux systems [SW05], for the time being we prefer to use FreeBSD for capturing high-volume streams.

4.2.7 Sensitivity to Programming Errors

A surprising consequence of operating a NIDS in a high-volume environment is the degree to which the environment exacerbates the effects of programming errors. While working on the Bro system, we have repeatedly encountered two kinds of mistakes that inevitably lead to significant problems no matter how minor they may appear at first: *(i)* memory leaks, and *(ii)* invalid assumptions about network data.

Even the smallest memory leak can drive the system to memory exhaustion. Therefore, we require that every function that is part of the system's main loop must not leak even a single byte. For example, we once introduced a small leak in Bro's code for determining whether a certain address is part of the local IP space. This bug caused an operational system with 1 GB of memory to crash after two hours. Unfortunately, these kinds of errors are particularly hard to find. With live traffic, the main indicator is that the system's memory consumption slowly increases over time. Yet this does not yield any hints about the culprit. Furthermore, memory leaks are often hard to reproduce using small captured traces. Yet on large traces, conventional memory checkers are terribly slow. In fact, such difficulties motivated us to instrument the Bro system to account for its memory consumption, as discussed in §4.3.1. (The common riposte that one should use a language with garbage collection, rather than C++, is not as simple as some might think, as garbage collection processing can lead to processing spikes similar to those discussed in §4.4.5.)

The second problem concerns invalid assumptions about the system's input. If a protocol decoder assumes network data to be in some particular format, it will eventually encounter some non-conforming input. This problem is exacerbated in high-volume environments, due to the traffic's diversity as well as its high rate, as discussed in §3.3.3. Indeed, we have several times encountered a protocol decoder running fine even on large traces, but crashing within seconds when deployed in one of our environments. Along these lines, not only does this observation mean that expecting strict conformance to an RFC will surely fail; but that expecting *any* sort of "reasonable" behavior risks failing. This problem is closely related to the observation of "crud" in network traffic (see §3.3.3) as well as "crash" attacks (see §2.4.3): not only must a NIDS be coded defensively to deal with bizarre-but-benign occurrences such as receivers acknowledging data that was never sent to them; but it must also be coded against the possibility of attackers maliciously sending ill-formed input in order to crash the NIDS, or, even worse, compromise it, as happened with the recent "Witty" worm [MS04].

4.3 Resource Measurement Methodology

To understand the memory and CPU usage of a NIDS, we need ways to measure them. In this section, we present the measurement methodology that we use in §4.4 and §4.5 to understand the memory usage of the Bro NIDS. We discuss *external tools*, which in general work with other systems as well; and *internal measurements* which we specifically implemented for the Bro system.

4.3.1 Memory Usage

If we want to understand the memory usage of a stateful NIDS, we need to track where exactly it stores the state. To analyze the memory layout during run-time we can either use external tools, or add internal measurement code.

External Tools

There are several tools available for memory debugging, of which we have used two that are freely available: *mpatrol* [MPa] and *valgrind* [Val]. The former comes as a library which is linked into the system and allows very fine-grained analysis by taking memory snapshots at user-controlled points of time. Unfortunately, *mpatrol* turned out to decrease the system's performance by multiple orders of magnitude, making it unusable on all but tiny traces (let alone real-time use). *Valgrind* takes another approach: it instruments the program on the instruction-level. While its performance is much better, it is still not sufficient for more than medium-size traces. Both programs proved to be most useful for finding illegal memory accesses.

Internal measurement

For internal measurements, we instrument the system using additional code to measure its current memory consumption. We identified Bro's main data structures and added methods to track their current size. During run-time, we regularly log these values. Additionally, we print the maximum *heap size* as reported by the system, and the *effective* memory allocation, i.e., the amount of memory currently handed out to the application by the C library's memory management functions. On Linux using *glibc* the heap size is monotonically increasing and provides us with an upper bound for the application's peak allocation.[5] We note that there is a gap between the peak heap size and the peak memory allocation: *glibc* keeps 8 bytes of hidden information in every allocated memory block, which is not counted against the current allocation. There is another pitfall when measuring memory. If we ask the C library for n bytes of memory, we may actually get $n+p$ with $p \geq 0$ padding bytes. For example, on Intel-Linux, *glibc's* malloc always aligns block sizes to multiples of eight and does not return less than 16 bytes. The memory allocation *includes* these padding bytes.

[5] Different systems behave differently. On FreeBSD, we can allow the C library to return unused memory to the system, thereby decreasing the total heap-size.

We note that in practice it is very hard to instrument a complex system accurately. Therefore, values delivered by internal instrumentation are often only approximations of lower bounds. Bro, e.g., creates data structures at many different locations and often recursively combines them into more complex structures. Often it is not determinable what part of the code should be held accountable for a particular chunk of memory (sometimes we cannot avoid counting allocations more than once). Consequently, we did not try to classify every single byte of allocated memory. Rather we identified the main contributors. By comparing their total to the current memory allocation, we ensure that we indeed correctly instrumented the code. In our experiments, on average we are able to classify about 90% of the memory allocation; the rest is allocated at locations that we did not instrument (we performed this verification on Linux systems; FreeBSD's C library does not provide an easy way to access the current effective allocation).

In the following sections, *total* memory allocation refers to the heap size. When we discuss the size of a particular data structure we refer to the values reported by our instrumentation, and thus to the lower bounds. These values *include* malloc's padding and assume a *glibc* based system. When we report the memory allocation for a particular trace, we always refer to the *maximum* for this trace.

4.3.2 CPU Usage

To measure the CPU usage of a NIDS, we have similar options as for quantifying memory usage: external tools and internal instrumentation.

External Tools

One obvious tool to measure CPU usage is the Unix *time* tool. It reports overall real-, user-, and system-time and does not impose any overhead on the observed process. It does not provide any hints about the system's real-time behavior, though (while the CPU load may be sufficiently low on average, processing spikes can lead to packet drops). Performance profilers, like *gprof* [GPr], provide fine-grained statistics, but their overhead is too large to infer real-time behavior.

Internal measurement

When examining CPU load, our main concern are packet drops. If we would know the exact time required to process each packet, we could predict when drops occur: assuming BPF's double buffering-scheme [AGJT03], we lose packets when the total time required to process the first buffer's packets exceeds the time for which packets can be stored in the second buffer.

Unfortunately, we cannot accurately measure the CPU time per packet. The overhead would be too large and the system's time granularity too coarse. Thus, we use another model. We measure the time t required for a group of n packets and chose n so that t lies in the order of the time resolution. When t exceeds the interval s in which the same n packets appeared on the network, we assume the system would drop packets. Additionally, assuming a packet buffer of size n, there will not be any packet drops as

long as t does *not* exceed s. We note that by averaging over n packets, we cannot blame a single packet or a small group of packets as being responsible for a sudden increase of CPU usage. Also, we cannot quantify how many packets would have been lost.

For our experiments, we used $n = 10,000$, resulting in times t within 30–50 ms with Bro's minimal configuration (on an Athlon XP 2600+). In §4.4.5 we see that this method is indeed able to identify processing spikes. Also, we see that fluctuations in per-packet processing times are easily observable.

We note that this is an idealized model. The system's time measurements are not accurate. Also, on a real system, there are other factors that influence the packet drop rate (such as load imposed by interrupts and other processes, or the OS itself). Finally, BPF's buffer implementation differs from our model. Thus, we do not claim to derive perfect values of real-time CPU usage. But our measurements give us some very valuable intuition regarding the system's behavior.

4.4 Analysis of Resource Usage

As discussed in §4.2.4, a high-performance network presents significant challenges with respect to resource demands. In this section we study the major resource management issues that we needed to address when deploying the Bro NIDS: state management that is either too liberal, or not existent at all as well as data and processing peaks causing missed packets. Most problems announced themselves either by exhausting the system's memory or by consuming all available CPU time—or both. But while the symptoms often are similar in appearance, they have a number of different causes. Often, detecting and fixing a particular problem leads to the immediate appearance of another one. Overall, each choice, e.g., analysis depth or parameter values, faces a trade-off between quality (i.e., detection rate) and quantity (i.e., required resources).

Next, we present the data on which our study is based. Then we discuss the management of two different kinds of state, and address packet drops due to network load as well as processing spikes. Finally, we recapitulate a recurring experience: a rather unusual trade-off between resource requirements and detection rate that ones faces in network intrusion detection.

4.4.1 Evaluation Data

While we gained our experiences and insights from deploying NIDSs operationally, live traffic poses limitations for any systematic performance evaluation study, e.g., in terms of repeatability of experiments. Therefore we draw upon a set of traces captured using tcpdump [TCP] at the MWN upstream (see §3.2.1) to demonstrate the challenges that a NIDS is facing:

- The trace `mwn-week-hdr` contains all TCP control packets (SYN, FIN, RESET) for a 6 day period. The compressed trace totals 73 GB, contains 365 M connections, and 1.2 G packets. 71% of the packets in the trace use port 80 (HTTP), with no other port comprising more than 3% of the traffic.

- `mwn-all-hdr` is a 2-hour trace containing all packet headers, captured during the daily "rush-hour" between 2 PM and 4 PM. (Basic statistics: 13 GB compressed, 471 M packets, 11 M TCP connections, 96.7% of the packets are TCP (57.8% HTTP, 3.7% FTP data transfer on port 20), 2.9% UDP).

- `mwn-cs-full` is a 2-hour trace including the full payload of all packets to/from one of the CS departments in MWN, with some high-volume servers excluded. This trace was captured at the same time as `mwn-all-hdr`. (11 GB compressed, 19 M packets, 404 K connections, 88% of the packets are TCP (41% HTTP, 11% NNTP), 10% UDP).

- `mwn-www-full` is a 2-hour trace including full payload from `dict.leo.org`. (2.8 GB compressed, 38 M packets, 1 M connections of which nearly all are HTTP).

- `mwn-irc-ddos` is a 2.5-day trace of `irc.leo.org` including full payload. During the monitoring period this IRC server was subjected to a large distributed denial-of-service attack which used random source addresses. The trace contains three major attack bursts, with peaks of 4,800, 5,300, and 35,000 packets per second, respectively (2.8 GB compressed, 76 M packets, 96% TCP / ports 6660–6668, 2% UDP).

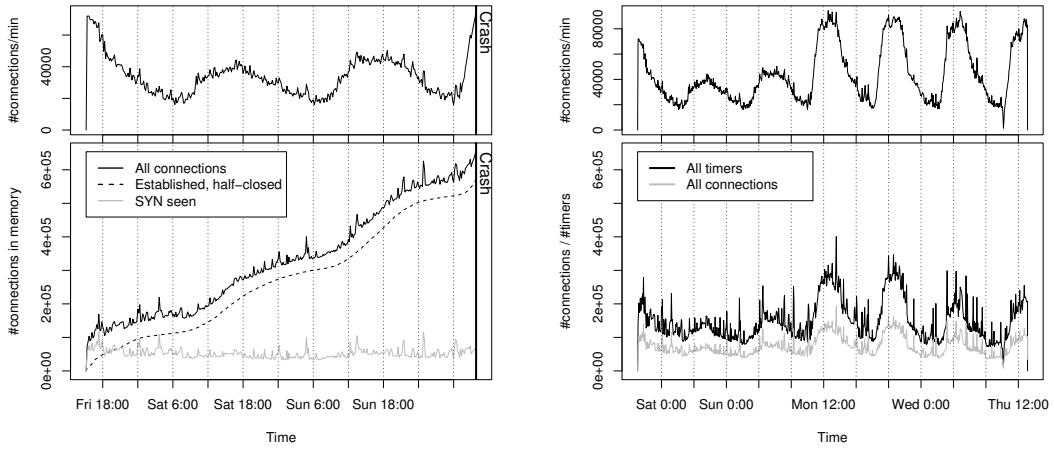
Tcpdump reported 0.01% or fewer lost packets for each of these traces. For the evaluation itself we had exclusive access to three systems. One was a Dual Athlon MP 1800+ with 2 GB Memory, running FreeBSD 5.2.1. We used this system mainly for any analysis involving the large `mwn-week-hdr` trace. The others were separate Athlon XP 2600+ based systems with 1 GB of RAM running Linux 2.4. To keep the analysis comparable in terms of memory use, we imposed a memory limit of 1 GB on all experiments independent of the system. Furthermore, results used for comparisons are derived using the same experimental system.

4.4.2 Connection State Management

For a stateful NIDS, it is vital to limit the overall memory requirements for state management to a tractable amount. In a high-volume environment, this is particularly difficult if the NIDS keeps per-connection state. In MWN, on a typical day we see up to 4,000 new TCP connections per second, and a total of about 75 M TCP connections per day.

The amount of memory required for connection state is determined by two factors: *(i)* the size of each state entry, and *(ii)* the maximum number of concurrent, still active connections. In Bro, the size of state entries differs due to factors such as IP defragmentation, TCP stream reassembly, and application-layer analysis, which determine the amount of associated state. To limit the number of concurrent connections, NIDSs employ timeouts to expire connection state, i.e., connections are removed from memory when some expected event (e.g., normal termination) has not happened for a (configurable) amount of time. In addition, some NIDS either limit the total number of concurrent connections or the amount of memory available for connection state. In

Figure 4.3 Connection state while processing the trace `mwn-week-hdr`.



(a) Default configuration.

(b) With inactivity timeouts.

either case, they flush connections aggressively once the limit is reached. Snort, for example, simply deletes five random connections once it reaches a configurable memory limit. While such a limit makes the memory requirements more predictable, it leads to ill-defined connection lifetimes.

For TCP connections, Bro's state entries consist of at least 240 bytes. If Bro activates an application analyzer for a connection, this can grow significantly. Running Bro in its default configuration[6] on the traces `mwn-week-hdr` and `mwn-www-full`, we observe average connection entry sizes of about 1 KB. When performing HTTP decoding on `mwn-www-full`, Bro needs 6–8 KB per connection (excluding data buffered in the stream reassembler, whose peak usage for this trace is in fact less than 5 KB in total).

When we store a significant number of bytes per connection, it is important to limit the number of concurrent connections. Yet, Figure 4.3(a)(bottom), shows that on `mwn-week-hdr`, with Bro's default configuration, the number of connections increases over time. Consequently, the system crashes after 2.5 days due to reaching the 1 GB memory limit.

The arrival of new connections (Figure 4.3(a)(top)) does not exhibit a similar increasing trend. This implies that the problem is not a surge in connection arrivals but rather Bro's state management. It does not limit the number of current connections, and at the same time it apparently fails to remove a sufficient number of connections from memory.

Since the number of connections for which only a single SYN packet has been seen does not increase dramatically, we conclude that the problem is not that Bro fails to

[6] If not stated otherwise, we deactivate most of Bro's analyzers for the measurements presented in this chapter. The only (major) user-level script we include is `conn.bro`, which outputs one-line summaries of all connections [Pax99]. In this configuration, Bro performs stateful analysis of all TCP control packets, but no application-layer analysis.

time out unsuccessful connection attempts. Indeed, Bro provides an explicit timeout mechanism for dealing with such connections. Decreasing these timeouts to the more aggressive thresholds used by Bro's `reduce-memory` configuration enables Bro to process an additional 110 minutes of the trace, only a minor gain.

Figure 4.3(a)(bottom) indicates that the number of connections in the established or half-closed state increases to the same degree as the total number of connections. After further analysis of Bro's state management, we find that many connections are not removed *at all*. This behavior is consistent with Bro's original design goal: to not impose limits that an attacker might exploit to evade detection [Pax99]. The problem faced by a NIDS is that there is no point at which it can be *sure* that a TCP connection in the established state can be safely removed. Thus, in accordance with its design goal, Bro does not remove connections unless it sees an indication that they are properly closed. There are at least three reasons for why one may not see the end of a connection: *(i)* hosts which, for whatever reason, do not close connections, *(ii)* packets missed by the NIDS itself (see §4.4.4 and §4.4.5), and *(iii)* artifacts caused by the monitoring environments (see §4.2.6).

Whatever the cause, however, accumulating connection state indefinitely over time is clearly not feasible. There are similar problems with UDP and ICMP connections. Most of these are also never removed from memory. While there already is a way to mitigate these problems[7], this still does not suffice. In §4.5.1 we develop an approach to mitigate this problem using inactivity timers. Figure 4.3(b) shows the number of connection entries in memory when using this extension. It clearly shows its success: Bro is now able to process the full trace `mwn-week-hdr`.

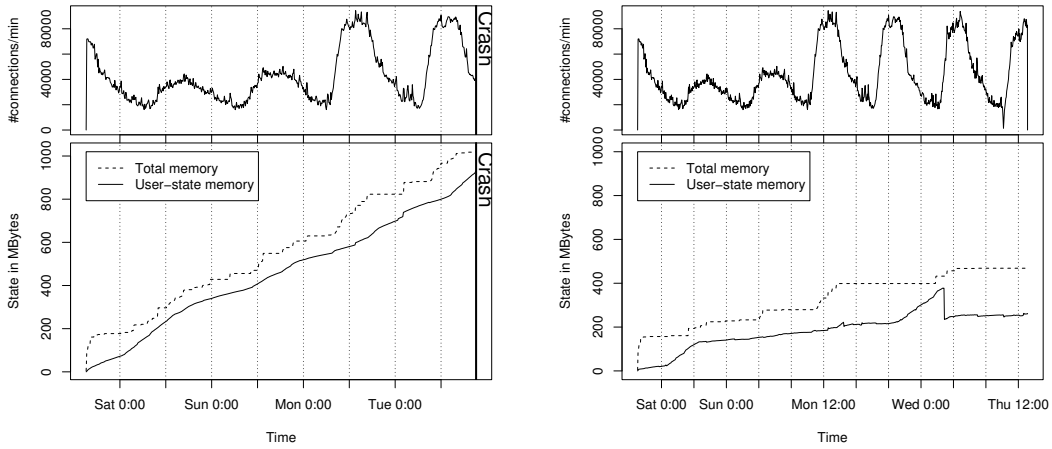
4.4.3 User-Level State Management

A NIDS may provide the users with the capability to dynamically create state themselves. While not all NIDSs provide such *user-level state*—Snort, for example, does not—other systems, like Bro, provide powerful scripting languages. But similar to the system's connection state, user-level state must be managed to avoid memory exhaustion. There are two approaches for doing so: *(i) implicit* state management, where the system automatically expires old state, perhaps using hints provided by the user; and *(ii) explicit* state management, where the user is responsible to flush the state at the right time.

Bro provides only explicit mechanisms, and the default policy scripts supplied with the public distribution make little use of these, motivated by Bro's original philosophy of retaining state as long as possible to resist evasion. Consequently, user-level state accumulates over time, generally causing the system to crash eventually. Two examples are the scan detector and the FTP analyzer. The former stores a table of host pairs for which communication was observed. Figure 4.4(a)(bottom) shows the memory allocation for the user-level state of the scan analyzer versus total memory allocation running on `mwn-week-hdr`. (The chosen configuration avoids the growth of connection state by

[7] There is a (by default deactivated) timeout to expire all not-further analyzed connections after a fixed amount of time.

Figure 4.4 Scan detector state while processing the trace `mwn-week-hdr`.



(a) Default configuration (plus inactivity timeouts for connections).

(b) With user-level timeouts.

using inactivity timeouts as described in §4.5.1.) Figure 4.4(a)(top) again shows the number of connections seen per minute. We recognize from Figure 4.4(a)(bottom) that the table mentioned above grows rapidly, since its entries are never removed. While this maximizes the scan detection rate (we will not miss any scans, thus thwarting evasion), it is infeasible in environments with large numbers of connections. For example on `mwn-week-hdr` the memory limit is reached after a bit more than 4 days. Here the main question is not whether to remove the table entries but *when*.

The FTP analyzer remembers which data-transfer connections have been negotiated via an FTP session's control channel, and removes this information only when the connection is indeed seen. While there is a point when this information can be safely removed—when the control connection terminates—this point is difficult to robustly detect from a user-level script, because Bro provides a multitude of event handlers for numerous kinds of connection termination. Even worse, there are (rare) cases when none of these events are generated.[8]

In §4.5.2 we develop another extension to Bro, user-level timeouts, for expiring table entries which results in a significantly reduced memory footprint as shown in Figure 4.4(b).

[8]The "crud" (see §3.3.3) seen in real-world networks sometimes misleads Bro's internal connection management. This also leads to some connections missing in Bro's connection summaries while others appear twice. We fixed this using the mechanism described in §4.5.2.

4.4.4 Packet Drops due to Network Load

In high-bandwidth environments, even after carefully tuning the NIDS to the traffic one still has to deal with inevitable system overloads. Given a heavily-loaded Gbps network, current PC hardware is not able to analyze every packet to the desired degree. For example, within MWN it is usually possible to generate Bro's connection summaries for all traffic, yet decoding and analyzing all traffic up to the HTTP protocol level is infeasible. To demonstrate how expensive detailed protocol analysis can be, we ran the HTTP analyzer on `mwn-www-full` and `mwn-cs-full`. Compared to generating connection summaries only, the total run-times increase by factors of 6.2 and 5.6, respectively.

Therefore, we need to find a subset of the traffic and types of analysis that the NIDS can handle given its limited CPU resources. Doing so for real-world traffic is especially challenging, as the traffic exhibits strong time-of-day and day-of-week effects. In the MWN, the traffic in the early afternoon is usually about 4 times larger than during the night; in fact, during night we *are* able to analyze all HTTP traffic. If we configure the analyzers for the most demanding times, we waste significant resources during low-volume intervals. Thus, we miss the opportunity to perform more detailed analysis during the off-hours. Alternatively, we could configure for the off-hours, but then we may suffer massive packet drops during the peaks.

In §4.5.3, we develop a mechanism to mitigate this problem by dynamically adjusting the NIDS to the current load. While this is extremely useful, we note that it remains an imperfect solution. We can still expect to encounter occasional peaks due to the variability of the traffic (see §3.3.3). Such situations can invalidate the assumptions underlying either the configuration of the NIDS or the processing of the NIDS itself. For example, the floods contained in `mwn-irc-ddos` contain millions of packets with essentially random TCP headers, which highly stress Bro's TCP state machine.

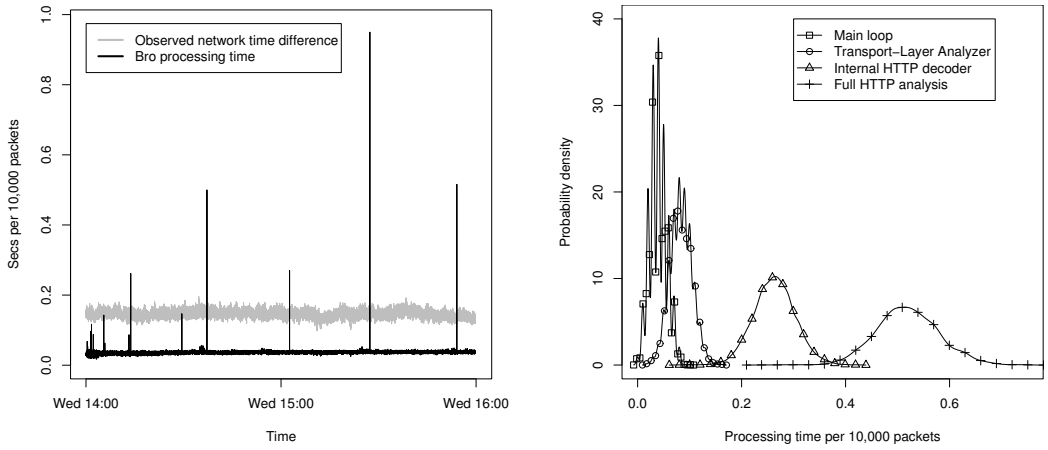
Thus, in practice, finding a configuration that never exceeds the resource constraints is next-to-impossible unless one keeps extremely large capacity margins. As perfect tuning is out of range within the trade-off of analysis depth vs. limited resources, we aim instead at a good balance: accepting some packet loss due to occasional overload situations while maintaining a reasonable analysis depth. For the MWN, we found a configuration which is able to run continuously (i.e., without the need to regularly checkpoint (§2.4.4) the system) even in such demanding situations as caused by floods or large-scale scans. The occurrences of packet drops is within acceptable limits (e.g., 2–3 times an hour).

4.4.5 Packet Drops due to Processing Spikes

A NIDS processing traffic in real-time has a limited per-packet processing budget. If it spends too much time on a single packet (or on a small bunch), it may miss subsequent ones. It turns out that the per-packet processing time fluctuates quite a bit. If these fluctuations together add up to a significant amount of CPU time, the system will inevitably drop packets.

We find that there are two major reasons for fluctuating packet processing times: First, a single packet can trigger a special, expensive type of processing. For example,

Figure 4.5 Processing time on the traces `mwn-all-hdr` (left) and `mwn-www-full` (right).



(a) Old hash table resizing spikes.

(b) Fluctuations in per-packets time.

Bro dynamically resizes its internal hash tables when their hash bucket chains exceed a certain average length, in order to ensure that lookups do not take too long. Figure 4.5(a) shows the processing time for each group of 10,000 packets and the timespan in which this group of 10,000 packets was observed on the network. The plot allows us to compare the time needed to process 10,000 packets vs. the time needed to transmit them across the monitored network link. If the processing is faster, indicating that Bro's processing is staying ahead, the corresponding black sample point is below the gray sample point. Otherwise, if the gray sample is below the black sample, Bro is unable to keep up with the incoming packet rate (see §4.3.2 for details about this measurement model).

Note the spikes in Bro's processing time. These are caused by hash table resizing. Each resize requires Bro to copy all pointers from the old table to a new position within the resized table. For large tables—such as those tracking 100s of thousands of connections—such a copy takes hundreds of msec. This time is alloted to a single packet and therefore to a single group of 10,000 packets, causing the spike in the processing time. Note that the spike exceeds the network time, indicating the danger of packet drops. We have verified that this phenomenon indeed leads to packet drops in our high-volume environments.

To address this problem, a colleague of us modified the hash table resizing to operate *incrementally*, i.e., per packet only a few entries are copied from the old table to the new table. Doing so distributes the resizing across multiple packets. While the amortized run-time of insert and remove operations on the table does not change, the worst-case run-time is decreased, which avoids excessive per-packet delays. We have confirmed that this change significantly reduces packet drops. Second, different types of packets require different kinds of analysis and therefore different processing times. For example, analyz-

ing TCP control packets requires less time than the application-layer analysis of HTTP data packets. Yet, since the content of packets differs even at the same processing levels, the times can vary significantly. Figure 4.5(b) shows the probability density functions of the processing time for groups of 10,000 packets for four different configurations. Each configuration adds an additional degree of analysis. The simplest configuration, "Main-loop," consists of Bro's main loop, which implements the full TCP state machine but does not generate any output. The second configuration, "Transport-Layer Analyzer," generates one-line summaries for every connection. The next configuration, "Internal HTTP decoder," does HTTP decoding without script-level analysis, while the last and most complex one, "Full HTTP analysis," adds script-level analysis. Note that the per-packet processing times vary significantly for each configuration. The amplitude of the fluctuations increases with the complexity of the configuration. This is due to the influence of the individual characteristics of each single packet, which gain more prominence as the depth of analysis increases. For the most complex configuration (full HTTP analysis), the standard deviation is 0.060 sec, whereas for the simplest configuration (only Bro's connection tracking and internal state management), the standard deviation is only 0.016 sec. In general, we observe that more detailed analysis increases the average processing time *and* increases its variability.

This increasing variability implies that interpreting such general statements as "decoding HTTP increases the run-time by x%" (cf. §4.4.4) need to be interpreted with caution. The actual change in run-time depends significantly on the particular input, and the additional processing delays may have even larger impact on real-time performance, by exceeding buffer capacities, than one might initially expect. More generally, this supports our discussion in §2.4.5 that judging NIDSs in simple terms such as maximum throughput (see, e.g., [HW02]) is indeed questionable.

4.4.6 Resources vs. Detection Rate

So far, we have seen several indications of a rather unusual trade-off in network intrusion detection: memory/CPU-time on one side against detection rate in the other. This is in contrast to computer science's more traditional trade-off between memory and CPU-time.

If we decrease the amount of state stored by the system, we automatically decrease the size of the internal data structures. Thus, we reduce both memory usage and processing time (even with efficient data structures like hash tables, more state requires more operations to maintain it). But, at the same time, we lose the ability to recognize attacks whose detection relies upon this state. Consider an interactive session in which the attacker first sends half of his attack, then waits some time before sending the remaining part. If the NIDS happens to remove the connection state before it has seen sufficient information to recognize the attack, it will fail to detect it. Similarly, if we decrease the CPU usage of the NIDS by avoiding certain kinds of analysis, we usually also reduce the amount of stored state. But again, we will now miss certain attacks.

Bro's original design emphasized detection. Many design decisions were taken to avoid false negatives, at the cost of large resource requirements. Unfortunately, as documented

above, this approach can be fatal when monitoring high-volume networks. For example, recall that by default Bro does not expire any UDP state. In terms of detection, this is correct: being a stateless protocol, there is no explicit time at which the state can be removed safely. On the other hand, keeping UDP state forever quickly exhausts all available memory on a high-volume link.

Trading resource usage against detection rate is an environment-specific policy decision. By leaving the final decision (e.g., choosing the concrete timeouts) to the user, one avoids predictability. As noted in §2.5.2, this is a variant of Kerkhoff's principle. We note that choosing appropriate timeouts is not easy.

4.5 High-Volume IDS Extensions

According to our observations in §4.4, there are two major areas where improvements of the NIDS show promise to improve high-volume intrusion detection: *(i)* state management and *(ii)* control of input volume. We devised new mechanisms for both of these. While their current implementation is naturally tied to Bro, the underlying ideas apply to other systems as well.

In the following, we discuss each improvement individually to gain an understanding of its impact. In practice, we use all of them. Together they are able to cope with the network load.

4.5.1 Connection State Management

One major contributor to the NIDS's memory requirements is the connection state (see §4.4). To reduce its volume, we use two complementary approaches: *(i)* introducing new timeouts to improve state expiration, and *(ii)* avoiding state creation whenever possible.

Inactivity Timeouts

In §4.4.2 we show that connections for which Bro does not see a regular termination accumulate. These amount to a significant share of the total connection state unless they are removed in some way. For expiring such connections, most NIDSs rely on an "inactivity timeout," i.e., they flush a connection's state if for some time no new activity is observed. There is one caveat: such a timeout relies on seeing all relevant packets. If a packet is missed, the NIDS might incorrectly assume that a connection is inactive. Missed packets can be related to drops due to monitoring issues (see §4.2.6, §4.4.4, and §4.4.5). But more importantly packets are also missed when the specified packet filter does not capture all relevant traffic. For example, if one only analyzes TCP SYN/FIN/RESET control packets, then an inactivity timeout degrades to a static maximum connection lifetime.

We added three inactivity timeouts to Bro, for TCP, UDP, and ICMP respectively, as these protocols show different characteristics. We also added the capability for the user's policy scripts to redefine these timeouts on a per-connection basis. The timeouts can be

adjusted separately based on the service/port number of the connection using a default policy script. This enables us to, for example, select shorter values for HTTP traffic than for SSH connections. Figure 4.3(b)(bottom) shows Bro's resource consumptions on `mwn-week-hdr` with an overall TCP inactivity timeout of 30 minutes.[9] In contrast to Figure 4.3(a)(bottom), we see that the number of concurrent connections in memory no longer exhibits the increasing trend. It instead follows the number of processed connections per time-interval closely (see Figure 4.3(b)(top)).

Naturally, inactivity timeouts should be as large as possible. But using timeouts in the order of tens of minutes or even hours revealed a significant problem with Bro's timer implementation: for processing efficiency, when a connection's state is removed, associated timers are only disabled, not removed. These timers are deleted once their original expiration time is reached. Using large timeout values, this results in more than 90% of the timers in memory being disabled. To reduce the memory requirements, we changed the code to explicitly remove old timers, expecting to accept a minor loss in performance. However, we found the run-time on `mwn-week-hdr` actually *decreased* by more than 20%. We suppose that this win is due to the reduced size of the data structure storing the timers. Figure 4.3(b)(bottom) shows the number of timers in memory after this change.

Connection Compressor

Examining the TCP connections monitored in our operational environments showed that a significant fraction corresponds to connection attempts without a reply. For example, for `mwn-all-hdr` 21% of all TCP connections are of this kind. For `mwn-week-hdr`, they account for 26% (recall that this trace contains only TCP control packets). Many of these connections are due to scans. In addition, we find that energetic flooding attacks—and also large worm events—vastly increase the number of connections attempts. Nearly all of these attempts are sure to fail.

As already discussed in §4.4.2 the minimum size of a connection state entry is 240 bytes. To reduce the memory requirements for connections that are never established, we implemented a *connection compressor* to compress their state. The idea behind the connection compressor is simple: defer the instantiation of full connection state until we see packets from both endpoints of a connection. As long as we only encounter packets from one endpoint, the compressor only keeps a *minimal state record*: fixed-size blocks of 36 bytes which contain just enough information to later instantiate the full state if required. Most notably, this minimal state contains the involved endpoints and some information from the initiating SYN packet (e.g., options, window size, and initial sequence number). If we do not see a reply after a configurable amount of time, the connection attempt is deemed unsuccessful, and its (minimal) state record is removed.

Using fixed size records allows for very efficient memory management: we simply allocate large memory chunks for storing the records and organize them in a FIFO. Since the FIFO ensures that connection attempts are ordered monotonically increasing

[9] The inactivity timer in this example degrades to a static maximum connection lifetime since `mwn-week-hdr` only contains TCP control packets.

in time, connection timeouts are extremely simple to implement. We just check if the first entry in the FIFO has expired. If so, we pop the record and continue until we reach a not yet expired entry.

Using the connection compressor when generating connection summaries[10] for `mwn-all-hdr` (`mwn-week-hdr`), the total connection state (including the minimal state records buffered inside the compressor) decreases by 51% (36%). In addition, we observed runtime benefits which at times can be rather significant. These benefits appear to depend on the traffic characteristics as well as the system's memory management, and merit further analysis to understand the detailed effects.

During our experiments, we encountered one problem when using the compressor: for some connections, Bro's interpretation of the connection's TCP state changes when activating the compressor. The compressor alters some corner-case facets of TCP state handling. While we attempt to model the original behavior as closely as possible, this is not always possible. In particular, we may see multiple packets from an originator before the responder answers. Sometimes originators send multiple *different* SYNs without waiting for a reply. In other cases we miss the start of a connection, stepping right into the data stream. While the change is definitely noticeable—affecting 2% of the connection summaries for `mwn-all-hdr`—nearly all of the disagreements are for connections which failed in some way. Since the semantics of not-well-formed connections are often ambiguous, these discrepancies are a minor cost if compared to the benefits of using the compressor.

Optionally, two more optimizations are possible. First, if the responder answers with a RESET to a connection request, the connection can be deleted immediately, rather than instantiated. In this case, the compressor avoids instantiating full connection state by directly reporting the rejected connection and flushing the minimal state record. This is particularly helpful during floods. Second—not yet implemented—we could choose to either not report non-established sessions at all, or only generate summaries such as "42 attempts from host a.b.c.d". For Bro, this would avoid creating user-level state for such attempts, potentially a significant savings.

4.5.2 User-Level State Management

As discussed in §4.4.3, a NIDS may provide the user with the capability to dynamically create their own custom state. We extended Bro's explicit state management to cope with the requirements of our high-volume environments by introducing an additional, implicit mechanism.

For Bro's existing, explicit state management mechanism, the fundamental (and only) question is *when* to decide to flush state. We inspected the state stored by its scripts and determined that a large fraction of the state is per-connection and stored in tables. Often, this can and should be removed when the connection terminates. To facilitate doing so, we added a new event which is reliably generated whenever a connection is removed

[10] For these measurements, we used inactivity timeouts of 30 minutes. We only analyzed TCP control packets.

from the system's state for whatever reason. We then modified the scripts to base their state management on the generation of this event (for example, we modified the FTP analyzer to remove state for tracking expected data-transfer connections whenever the corresponding control session terminates).

Often, however, we would rather have implicit—i.e., automatic—state management, relieving us of the responsibility to explicitly remove the state. Carefully-designed timeouts provide a general means for doing so. While Bro supports user-configurable timers, using them for state management requires the user to manually install the timers and also specify handlers to invoke when the timers expire.

To support implicit state management, we extended Bro's table and set data structures to support per-element timeouts of three different flavors: creation, write, and read. That is, for a given table (or set), the user can specify in the policy script a timeout associated with each element which expires T seconds after any of: the element's creation; the last time the element was updated; or the last time the element was accessed.

One benefit of this approach is that these timeouts provide a simple but effective way to add state management to already-existing scripts. Consider for example the scan analyzer, which, as mentioned above, can consume a great deal of state. Figure 4.4(b)(bottom), shows the quite significant effects of running Bro on `mwn-week-hdr` using 15-minute read timeouts for the scan detection tables. (Note, the large spike on Wednesday seems to stem from a single host in the scan-detection data structures that performs large vertical scans. Eventually, all of its state is expired at once.)

Adding timeouts to the scan detector also revealed a problem, though: sometimes state does not exist in isolation, but in context with other entities. In this case, when we implicitly remove it, we can introduce inconsistencies. For example, the scan detector contains one table tracking pairs of hosts and another counting for each host the number of such distinct pairs involving the host. Automatically removing an entry from the first table invalidates the counter in the second.

To accommodate such relationships, we added an additional attribute to Bro's table type which specifies a script function to call whenever one of the table's entries expires and is removed. In the scan detector, this functions simply adjusts the counter and thus maintains consistency between the two tables.

4.5.3 Dynamically Controlling Packet Load

As discussed in §4.4.4, to avoid CPU exhaustion we need to find ways to control the packet load. Doing so statically—i.e., by controlling the BPF filter Bro uses for its packet capture—lacks the flexibility necessary to adapt to the wide range of conditions we can encounter over a relatively short period of time. Thus, we devised two new dynamic mechanisms: *(i) load-levels*, which allow us to adapt to the system's current load, and *(ii)* a *flood detector*.

We define *load-levels* as a set of packet filters for which we maintain an ordering. Each filter that is "larger" in the ordering than another imposes a greater load on the NIDS than its predecessor. (Note that the extra load is not primarily due to the burden of

the packet filtering per se, but rather the associated application analyzers that become active due to the packets captured by the filter, and the processing of the events that these analyzers then generate.)

At any time, the kernel has exactly one of these filters installed. By continuously monitoring its own performance, the NIDS tries to detect overloads (ideally, incipient ones) and idle times. During overloads, it backs off to a filter earlier in the ordering (i.e., one requiring less processing); during idle times, it ramps up to a filter that reflects more processing, because during these times the NIDS has sufficient CPU resources available and can afford to do so.

The filters are defined by means of Bro's scripting language. For example, the following code makes the activation of the DNS, SMTP and FTP decoders dynamic rather than static. A decoder is enabled if the system's level is less than or equal to the specified load level:

```
redef capture_load_levels += {
        ["dns"]   = LoadLevel1,
        ["smtp"]  = LoadLevel2,
        ["ftp"]   = LoadLevel3,
};
```

For such an adaptive scheme to work—particularly given its feedback nature—it is important to estimate load correctly to avoid rapid oscillations. Two of the possible metrics are CPU utilization and the presence of packet drops. (For either, we would generally average the corresponding values over say a couple of minutes, to avoid overresponding to short-term fluctuations.)

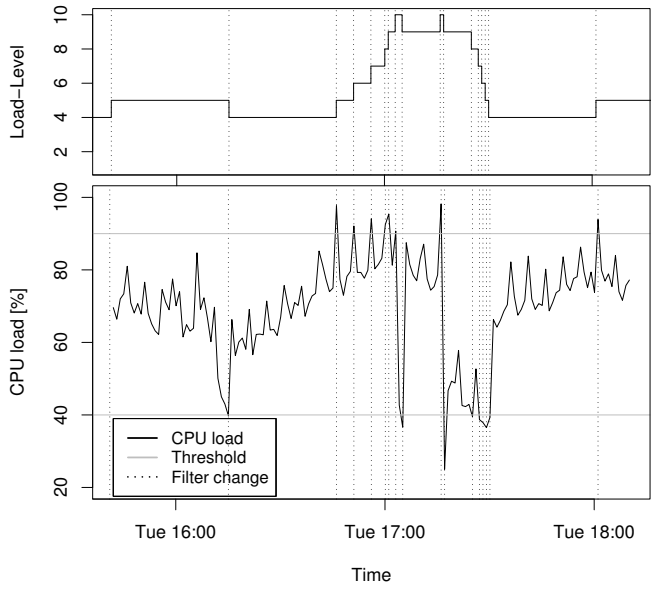
We have experimented with both of these metrics. We found that while the latter (packet drops) is indeed prone to oscillations, the former (CPU utilization) proves to work well in practice. The particular algorithm we settled on is to adjust the current packet filter if the CPU utilization averaged over two minutes is either *(i)* above 90% or *(ii)* below 40%, respectively.

To make this work, we cannot afford to compile new BPF filters whenever we adapt. Accordingly, we precompile the entire set of filters to keep switching inexpensive in terms of CPU usage. As FreeBSD's packet filter flushes its captured-traffic buffers when installing a new filter, we also devised a patch for the kernel-level driver to avoid losing packets.

Figure 4.6 shows an example of load-levels used operationally at MWN. When the CPU load crosses the upper threshold (Figure 4.6, bottom), the current load-level increases, i.e., Bro shifts to a more restrictive filter (Figure 4.6, top). Accordingly, if the load falls below the lower threshold, a more permissive filter is installed.

The MWN environment includes an IRC server which, unfortunately, is a regular victim of denial-of-service floods. It sometimes suffers such attacks several times a week, being at times targeted by more than 35,000 packets per second. Such a flood puts an immense load on a stateful NIDS, although ideally the NIDS should just ignore the attack traffic (of course after logging the fact), since there's generally no deeper semantic meaning to it other than clogging a resource by sheer brute force.

Figure 4.6 Effect of load-levels.



None of the mechanisms discussed above can accommodate in a "reasonable" fashion the flood present in `mwn-irc-ddos`. Thus, we devised a *flood detector* that is able to recognize floods and dynamically installs a new filter until the attack passes.

Detecting floods is straightforward: count the number of new connections per local host and assume a flood to be in progress if the count surpasses a (customizable) threshold. Doing so requires keeping an additional counter per internal host, which can be quite expensive. Therefore, we instead sample connection attempts. Similarly, instead of ignoring all packets to the victim after detecting a flood, we sample them at a low rate to quickly detect the end of the flood.

For Bro, we added such a flood detector by means of a new script. It samples connection attempts at a (customizable) rate of 1 out of 100, reporting a flood if the estimated number of new connections per minute exceeds a threshold (default, 30,000). When this occurs, the script installs a host filter, sampling packets also at 1:100.[11]

On `mwn-irc-ddos`, this mechanism detects all contained attack bursts. The total memory allocation stays below 122 M (whereas all other considered configurations exhaust the memory limit of 1 GB during the last attack burst, at the latest).

[11] Unfortunately, the standard BPF packet filter does not support sampling. Thus, we augmented Bro's packet capture with a new, user-level packet filter that can directly support sampling. While this does not relieve the main process from receiving the flood's packets, they do not reach the system's main loop.

4.5.4 Using High-Performance Monitoring Hardware

In high-performance environments, commodity hardware is running at its limits. As discussed in §2.3.10, a common approach to push the limit further is to supplement off-the-shelf hardware with special-purpose components. In this spirit, we interfaced a *Endace DAG* [End] network monitoring card to the Bro system.

The DAG series is specifically built for high-performance packet capturing. These cards directly transfer captured network packets into a host's memory at wire-speed, guaranteeing not to drop any packets as long as the host is able to process the packets quickly enough. The cards provide accurate packet timestamps with a resolution in the order of tens of nanoseconds. Optionally, DAG cards can be equipped with an additional co-processor to pre-process packets on the card. For a NIDS, there are three advantages of using such a card rather than a standard PC network card:

CPU-Offloading. Capturing with DAG cards does essentially not impose any load on the host CPU as the packets are directly transfered into memory. With a conventional card, packets are copied several times on their way from the card's buffer to the user-space memory. This process is expensive in terms of CPU cycles and may also incur a significant interrupt-load on the system.

Improved Reliability. By design, no packets are lost on a DAG card. With standard cards, there is no such guarantee.

On-Card Filtering. DAG cards can pre-process packets using either the on-board FPGA or the optional co-processor. In particular, they can *filter* packets, i.e., immediately discard any packets that the NIDS is not interested in anyway. With conventional cards, this filtering takes place inside the host's kernel which significantly increases the system's CPU and interrupt loads.

To make these benefits available to the Bro NIDS, we extended the system to directly access a DAG card's native programming interface. As Bro used to support only libpcap-based capturing, this required some internal changes.[12] We do not yet leverage the DAG card for filtering packets. Bro depends on BPF-style filter expressions to select a subset of packets for its analysis. For the DAG cards, there already exists a prototypical implementation of an on-board BPF-machine. As soon as this is ready for production use, we will adapt Bro to push the filter onto the card which requires only a minor modification. As until then we cannot exploit the full potential of the cards, so far we have only performed informal performance measurements, using the DAG 4.3GE card installed in one of our MWN monitoring systems (see §3.2.1). These experiments indicate that a DAG card provides significant benefits in terms of reliability and system load.

[12] Endace also provides a libpcap-based interface to their DAG cards. This introduces an additional layer of processing though which we prefer to avoid.

4.6 Summary

In large-scale environments, network intrusion detection systems face extreme challenges with respect to traffic volume, traffic diversity, and resource management. In this chapter, we first discussed our operational experiences with different NIDSs. Then we analyzed the resource demands of one system in depth, improving its resource management as necessary for successful deployment in our environments.

We deployed four NIDSs: Snort, Bro, Dragon, and IntruShield. While in their default configurations, Snort, Dragon, and IntruShield tend to generate lots of (false) alerts, Bro is relatively quite as it does not deploy a signature library by default. Tuning any of the systems is difficult and time-consuming, with Bro providing the most flexibility but also requiring the most expertise. Snort's signatures exhibit a general lack of quality; the Dragon and IntruShield signatures appear to be more reliable. The user interface of Snort and Bro consists of ASCII-files for configuration and output. Dragon and IntruShield ship with graphical interfaces, with Dragon also providing command-line access. Both graphical user interfaces show significant limitations: they are very basic in nature, lack responsiveness, and exhibit technical problems. In terms of comparing the detection rate of NIDSs, we observe that it is hard to achieve a fair picture: differing internal semantics often lead to comparing "apples and oranges". Independent of any particular NIDS, traffic of high-performance environments demands a great deal of the whole monitoring environment (e.g., the monitoring router and the OS-level packet-capture subsystem). From a developer's perspective, when implementing code for a NIDS, it is important to remember that even small programming errors, like tiny memory leaks or not fully validated input, will almost certainly cause problems eventually.

We took a close a look at the resource usage of the Bro system. We identified the main contributors to CPU load and memory usage, understood the trade-offs involved when tuning the system to alleviate their impact, and devised new mechanisms when existing tuning parameters did not suffice. While the symptoms of resource exhaustion often appeared similar, the problems were due to a number of different reasons. First, the Bro system's state management was designed to resist evasion, and thus traded detection-rate in favor of resource consumption. Second, the dynamic nature of the traffic makes it hard to find a stable point of operation without wasting resources during idle times, affecting both long-term traffic variations (e.g., due to strong time-of-day effects) and short-term fluctuations (e.g., due to "heavy-tailed" traffic and varying packet processing times). For problems that could not be solved with the available tuning parameters, we developed new mechanisms. We improved state management by introducing new time-outs, deferring instantiation of connection state by means of a *connection compressor*, and adding new means to dynamically control the packet load (*load-levels* to automatically adapt the NIDS to the current network load; a *flood detector* to revert to sampling of high-volume denial-of-service attack flows).

One of our most important observations is that network intrusion detection faces a rather unusual trade-off between CPU/memory demands and detection rate. Reducing the state kept by a system decreases both the CPU load and the memory footprint. Yet, these savings come at the cost of an increased probability to miss attacks if the system

requires the state for reliable detection. On the other hand, increasing the detection rate by keeping more state implies higher resource demands. A similar reasoning holds in terms of selecting the depth of the NIDS's analysis. Choosing this trade-off is an environment-specific decision and thus part of the policy.

In summary, this work provides us with *(i)* a thorough understanding of the trade-offs involved when deploying a NIDS for use in a high-volume network, and *(ii)* the tuning mechanisms necessary to successfully operate these systems in such challenging environments.

5 Separating State From Processing

He who knows only his own side of the case, knows little of that.

— *John Stuart Mill*

Network intrusion detection systems critically rely on processing a great amount of state. Often, much of this state resides solely in the volatile processor memory accessible to a single user-level process on a single machine. In this chapter we highlight the power of *independent state*, i.e., internal fine-grained state that can be propagated from one instance of a NIDS to others running either concurrently or subsequently. Independent state provides us with a wealth of possible applications that hold promise for enhancing the capabilities of NIDSs. We discuss an implementation of independent state for the Bro NIDS and examine how we can leverage independent state for distributed processing, load parallelization, selective preservation of state across restarts and crashes, dynamic reconfiguration, high-level policy maintenance, support for profiling and debugging, and inclusion of host-based state. We evaluate the performance of our implementation and show that it is indeed suitable for use in high-performance networks.

5.1 Overview

As we see in the previous chapter, a NIDS relies on managing a significant amount of state. The state reflects the NIDS's model of the communications currently active in the network; it supports the detection by providing context. Managing this state raises significant issues though. In §4, the sheer volume required us to develop expiration heuristics to expunge state aggressively. Another orthogonal approach to reduce the load on an individual NIDS is splitting the analysis across multiple instances of the system. Such a distribution multiplies the available resources and thus allows to keep more state in total, thereby enabling a deeper inspection.

In such a distributed setup, each process only performs a part of the overall detection. Hence, its state also reflects only a part of the whole picture, i.e., the network context that is available to support its decisions is significantly reduced. To mitigate this loss, many NIDSs include communication mechanisms to share state between instances. However, usually the state that is exchanged is only a minor subset of the NIDS's full state; often it is confined to high-level facts such as alerts. The much richer and bulkier internal state of the NIDS (e.g., all the user-level state; see §4.4.3), remains exactly that, internal. It cannot be accessed by other processes, and it is permanently lost upon termination of the NIDS (which, due to a crash, may happen unexpectedly). Therefore, even with communication in place, the instances are still lacking a major fraction of the whole picture.

In this chapter, we develop the concept of *independent state*, i.e., internal *fine-grained* state that can be propagated from one instance of a NIDS to others. Our goal is to enable much of the semantically rich, detailed state that hitherto could exist only within a single executing process to become independent of that process. We consider two basic types of independent state. *Spatially independent* state can be propagated from one instance of a NIDS to other concurrently executing instances. *Temporally independent* state continues to exist after an instance (or all instances) of a NIDS has (or have) exited. For both types of independence, the state essentially exists "outside" of any particular process.

We implemented the concept of independent state within the Bro NIDS. For this system, we developed a framework which is

Transparent. From the user's point of view, independent state is processed in the same way as non-independent state. All the system needs to know is which state is to be independent.

Unified. Internally, all of the systems' state is covered in the same way. The framework provides the mechanisms to make arbitrary state independent.

Efficient. Our implementation of the independent state concept is suitable for use in high-performance environments.

Having such a framework, distributed processing is easy to setup. Consider a set of NIDSs at different locations of a network, each able to identify suspicious activity in its segment. Traditionally, either each NIDS works independently of its peers, or there is an explicit mechanism to send, receive and incorporate alerts. With independent state, it is possible to transparently leverage the others' results. We simply tell the systems what state should be synchronized among them.

Independent state provides us with a wealth of other possible applications, including various ways of load-balancing; selectively preserving key state across restarts and crashes; dynamically reconfiguring the operation of the NIDS on-the-fly; tracking the usage of user-level state over time to support high-level policy maintenance; and enabling detailed profiling and debugging. We implemented all of these and discuss them in depth after presenting the framework.

Independent state is not restricted to being shared between NIDSs. Generally, *any* device can participate in the exchange, and we demonstrate the power of this approach by including *host-based* state into the analysis. As a case study, we interfaced the Apache Web server with the Bro system, and evaluated this combo in a testbed as well as in two production environments.

In the next section, §5.2, we introduce some terminology. In §5.3, we identify types of NIDS state. Then we discuss our independent state architecture in §5.4. We examine in §5.5 the powerful features and applications mentioned above that fine-grained independent state enables. In §5.6 we evaluate the communication performance of the architecture. We include host-based state into the analysis in §5.7. Finally, we summarize in §5.8.

5.2 Terminology

First, we introduce some terminology. In general, we can distinguish between several types of state. We term state that can be shared between multiple processes *independent state*. There are two types of independent state: *spatially independent* state may be shared by *concurrently* running processes; *temporally independent* state may be shared by *subsequently* running processes. State may be both spatially independent and temporally independent at the same time.

We refer to state that does not change over the course of the execution of a NIDS instance as *static state*. Such state often includes the NIDS's configuration (e.g., the signature set it uses) and perhaps a database of information about the network it is protecting, such as the types of operating systems installed on the monitored hosts or their "active mapping" profiles (see §2.4.3).[1] We refer to NIDS state that does change, on the other hand, as *dynamic state*. Dynamic state includes, e.g., connection-state and user-level state. For dynamic state, we make a further distinction: we refer to dynamic state that effectively exists at only a single unit (quantum) of time as *volatile state*, and state that exists over a longer interval of time as *non-volatile state*. For the Bro system, examples of volatile state include events generated by the event engine, and the values of local variables when Bro invokes policy script functions (since Bro specifies the execution of such functions as atomic, i.e., they run to completion before Bro advances its clock by processing the next packet).

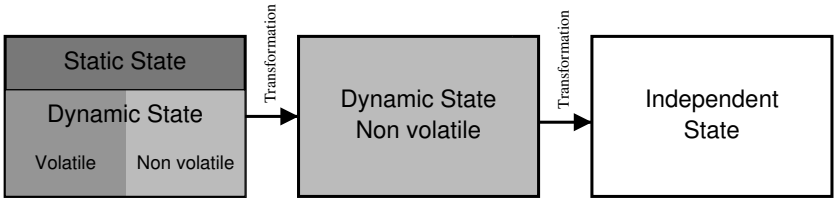
5.3 Bro's State

To put the concept of independent state into practice, our first task is to identify the different types of state present in a NIDS. While there is a common subset of state held by most NIDSs (e.g., connection state), in this section we delve into Bro's particulars to make the concepts concrete. Recall that for Bro, there are two main layers of operation, each of which stores a significant amount of state. The event engine layer produces a stream of events reflecting the activity present in the traffic stream. While the event engine's operation is tunable by redefining user-visible parameters, its algorithms—and therefore the types of state it stores—are fixed. On the other hand, the policy script layer allows the user to arbitrarily change and extend the standard set of scripts. Since this layer equips the user with a full scripting language, the corresponding types of state are only determined when the scripts are loaded at run-time.

The ultimate goal for our architecture is to support making all of Bro's state both temporally and spatially independent. Our first observation is that this should be easy for static state. Since, by definition, static state cannot change over time, there must already be a means to specify it upon start-up, which we can use again to recreate the state in other instances. However, we also develop the theme that converting static state to dynamic state extends the flexibility of a NIDS by enabling modifications during run-

[1] We note that the latter might in fact change over the course of execution, but if the NIDS does not have a means to incorporate such changes, the state is effectively static.

Figure 5.1 Transformation of state.



time. Therefore, our architecture aims at accommodating this. Our second observation is that volatile state, by definition, cannot be independent. Hence, we also aim to find ways to convert volatile state to non-volatile state to enable making it independent. In summary, our goal is to transform as much state as possible to being dynamic non-volatile, and thus temporally- and spatially-independent; see Figure 5.1. Next, we discuss the types of state that Bro stores in its two main layers. We note that we consider events as user-level state because their role is to trigger script-level actions. The event engine does not itself process events, it only generates them.

5.3.1 Event Engine State

We identified five main types of state internal to Bro's event-engine: connection state, analyzer state, timers, control state, and packets. Using our terminology, all but packets are dynamic non-volatile.

Connection State. Bro's main unit of analysis is a connection. For each connection, it stores a variety of information such as its start time, the hosts (IP addresses) involved, the amount of data transferred so far, transport protocol state, etc. In terms of volume, the per-connection state is by far the most dominant internal state (see §4.4.2).

Analyzer State. Most network protocols are stateful. Therefore, Bro's protocol-specific analyzers need to maintain state to remember the current state of a connection's protocol dialog. For example, the TCP analyzer buffers out-of-sequence data, and the HTTP analyzer accumulates client and server headers. The policy-neutral components of the analyzers, which are located inside the event engine, attach their state to Bro's generic connection state, leveraging its state expiration. Some analyzers also store global data inside the event engine which is not linked to a particular connection. One example is the stepping stone analyzer [ZP00b], which keeps a set of candidate stepping stone connections.

Timers. A timer schedules actions for points of time in the future. Timers are Bro's main mechanism for expiring state. For example, Bro installs a timer for every connection attempt it sees. When such a timer expires (usually after a couple of seconds), Bro removes the connection's state if it has not successfully been

established. The number of concurrently active timers can easily reach tens of thousands on a high-volume link (see §4.5.1).

Control State. Control state is parameters that impact the operation of the event engine. While they are small in terms of volume, they have a major effect on the performance of the system. The most significant of these parameters are timeout values and the current packet filter which implicitly controls which analyzers execute. User scripts can dynamically change the parameters.

Packets. A network packet is the most basic unit of data processed by Bro. In terms of state, packets are dynamic volatile by definition since Bro's notion of time is derived from them.

5.3.2 User-Level State

As discussed in §4.4.3, Bro allows users to create user-level state by manipulating policy script variables. We now extend this notion of user-level state to include other types of state at the policy-layer. In total, we have identified six types of user-level state:

Scripts. Scripts are programs written in Bro's scripting language. The language provides the user with typical features of other programming languages (e.g., flow control statements, associative tables, compound types, type-safety, function calls) as well as domain-specific elements (e.g., event handlers to react to events, specific data types such a host addresses). On startup, Bro loads a set of scripts which define its run-time behavior. The scripts are contained in ASCII-files and hence static state.

Data. Script-level variables store dynamic state which is maintained by script functions and event handlers when they are executed. Such *data* can be stored either in global variables (then it is non-volatile) or in variables local to the function/handler (then it is volatile since Bro fully executes function calls within a single unit of time). The state that is accumulated by scripts includes the state of the corresponding policy-specific components of the analyzers.

Variables are of one of a number of different data types. Tables indexed by a set of types and yielding an arbitrary type are particularly common. For example, Bro's scan detector—which detects horizontal and vertical scanning, as well as password guessing—keeps *(i)* a large script-level table containing pairs of communicating hosts, *(ii)* another table counting to how many different hosts a particular host has attempted to initiate connections, and *(iii)* in fact 21 additional smaller tables and sets (as of Bro version 0.8a39). For automatic expiration, Bro keeps timestamps for the entries in tables (and sets) which drive timeouts used to delete the entries (see §4.5.2).

Data Operations. A script-level variable can be modified by different kinds of script operations, depending on the type of variable. For example, for tables one can insert or remove an entry. Ordinarily, operations would be viewed not as state but as

transformations of state. However, with our definition of volatile state, we can view specific instances of operations as existing momentarily, and thus constituting dynamic volatile state. Designating them as such then allows us to transform them into non-volatile state to, e.g., produce logs of operations.

Events. Events are raised by the event-engine or by script statements. They trigger the execution of the corresponding event handlers. An event consists of the name of the event as well as the arguments (and their types) which are passed to the event handlers. Similar to data operations, we view the generation of an event as a form of dynamic volatile state.

Function Calls. Function calls are executed by event handlers or other script functions. Similar to events, a function call consists of the name of the function and instantiations of the parameters. Likewise, we view function calls as dynamic volatile state. The principle difference to events is that function calls return values, while event handler invocations do not.

Signatures. Signatures are, essentially, a set of byte-level patterns. In Bro, they are expressed by means of a dedicated signature language which is independent of the scripting language (see §6). These signatures are static state. For some NIDSs (e.g., Snort), signatures comprise the main type of user-level state.

5.4 Architecture for Independent State

Our architecture for independent state within the Bro NIDS is composed of three major building-blocks which we discuss in the following. First, we present our main tool for implementing independent-state: a serialization framework. We then turn to how the user interacts with the framework by discussing its script-level interface, and we address the need for secure and robust communication between concurrent NIDS processes. Finally, we also discuss how external applications can partake in the sharing of state.

5.4.1 Serialization

The main mechanism for making Bro's state independent is a *serialization* framework that enables us to convert all of Bro's state into a self-contained binary representation and back. Given this framework, we can, for example, make state temporally independent by serializing it into a file at the termination of a Bro instance. A new instance can then read it back upon start-up. Similarly, to make state spatially independent, we can send it over the network to some remote instance. However, adding full serializability to a complex system like Bro, which was not designed with this in mind, raises numerous subtle issues we must address which we discuss in this section.

Making object-oriented data structures serializable is, by itself, fairly straightforward and well-established practice [Sou94]. Each class gets two new methods, one for serializing and one for de-serializing. When called, each of them first calls the corresponding

method of the class's ancestor, and then (de-)serializes all attributes unique to the class itself. We simply followed this approach, using a generic class as the interface to convert between internal data types and an external representation. Doing so keeps the serialization process independent of the underlying external format, so we are, for example, able to create both a (portable) binary representation as well as an one in XML.

For memory management, Bro uses reference counting extensively. When de-serializing we need to recreate the reference structure. To this end, objects are assigned a set of unique identifiers. Each object is fully serialized only once, when encountered for the first time. Upon subsequent serialization requests, e.g., due to being referenced by some other object, we only output its ID if the object is still unchanged.

A basic problem that arises is the time needed to serialize state. Bro is a real-time system that must keep up with a high-volume stream of packets. If it spends too much time on other things than processing packets, it risks dropping packets (see §4.4.4). Therefore, we implemented *incremental* serialization: serialization proceeds in steps intermixed with packet processing. While this requires more time to finish the serialization, our ability to keep pace with the packet stream improves.

Incremental serialization leads to another problem though. By serializing chunks of the state at different points of time, we may incur inconsistencies. Consider, for example, two script-level tables that contain related data derived from the same connection. It may happen that after the first table has been serialized, we process some packets that remove the connection's data from the second table. Thus, serializing it in the next step would leave the two tables out of synchronization. To address this problem, we use a transaction model [TS02]. While an incremental serialization is in progress, we internally funnel all state-changing data operations through a logging component. By converting the operations into non-volatile state (see the discussion of serializing data operations below), we produce a transaction log which is appended to the serialization once it is finished. This log is then replayed upon reinstantiation to ensure consistency of the state.

We now turn to looking at the different types of state, discussing some of the particulars involved in their serialization. We also mention some points for which we have deliberately skipped the implementation for now. For some of these, it is not in fact clear whether supporting them makes sense. For others, adding support is straightforward, but to date we have not found a need in operational use. We will add them in the future if the need arises.

Serialization of Event Engine State

We now discuss how we serialize the types of event-engine state that we identify in §5.3.1.

Connection and Analyzer State. Connection state and the attached analyzer state is straight-forward to serialize and to restore. The main problem is the potentially very large number of concurrent active connections. For example, on the high-performance links discussed in §3, we regularly encounter more than 30,000 concurrent connections, even when using aggressively small timeouts.

This leads to two major problems. First, the volume of the data is high: due to the attached analyzer state, a single connection entry can exceed 1 KB in-memory, and the external representation—which we have not optimized for space—is even larger. Second, and more importantly, it simply does not make sense to serialize *all* connections: the vast majority are quite short (100s of msec or a few seconds). Considering that it takes some time to serialize them (particularly using incremental serialization as described above), most of them will already be finished before they are de-serialized. Therefore, we only selectively serialize connections specifically requested by the user. For example, the user can restrict state-independence to types of connections expected to be long-lived, like FTP or SSH connections.

As a consequence, we have not yet implemented serialization for all analyzer-specific connection state. While all transport-layer protocols are fully serializable, so far the only supported application-layer protocol is FTP. But adding serializability to other protocols as needed should be straightforward.[2] Similarly, while we have not yet implemented serialization for any of the analyzers keeping global state (in contrast to attaching their state to connection entries), this will be easy to achieve if the need arises.

Timers. Independent instances of a NIDS have different notions of time; even when running concurrently, a tight time synchronization is hard to achieve. Therefore, when de-serializing timers, we need to reschedule appropriately. We reschedule timers according to the time left until expiration when serializing them. This ensures that the total time interval covered by a timer is the one specified upon instantiation. We note that with these semantics, exchanging timers makes most sense for spatially independent state. So far, we only implemented serialization for timers related to Bro's connection management, but supporting the other types will be straight-forward when needed.

Control State. Unfortunately, Bro's control state is not located in some well-defined class but distributed across several places. User-redefinable parameters are implemented via script-level variables. Thus, we leverage their serialization as described below. For the packet filter, we added a method to output a string containing the filter specification and then read it back and change the filter to reflect it. We note that the system may need some time to compile and install a new specification; usually a couple of milliseconds. Additionally, at least on FreeBSD, it clears the current packet buffers (if not patched as discussed in §4.5.3). As therefore it is likely that we miss some packets, there is a trade-off involved when installing a new filter.

Packets. Packets are straight-forward to serialize as they are fully self-contained units of data. The only problem is that of the packets' timestamps: as system clocks are usually not synchronized among independent instances, a packet's original timestamp is usually meaningless to the receiving instance. Moreover, as packets

[2] For SSH, there is not any application-layer analyzer state; due to the encrypted nature of the payload, Bro presently only decodes the initial handshake and then confines itself to the generic TCP analysis.

drive Bro's internal clock, using the original timestamp may cause the clock to jump back and forth—which would violate one of the system's internal assumptions. Thus, when de-serializing, we replace the packet's original time-stamp with Bro's current time. However, in the future, we will extend Bro's concept of time to cope with non-monotonic timestamps.

Serialization of User-Level State

Next, we discuss serialization of user-level state as identified in §5.3.2.

Scripts. The main components of scripts are type definitions, global variables, functions, and event handlers. All of them are fully serializable and de-serializable. In addition, we added support for changing definitions and adding new elements during run-time. Thus, this type of state is no longer static but dynamic.

Data. We added serialization to all of Bro's types of values which can be bound to variables, with full type-checking during reinstantiation. While not articulated as such in Bro's design, we identified two basic types of values, static and mutable. Static values (e.g., integers) are not themselves changeable at the script-level, since they are deep-copied upon assignment. Mutable values are container values (e.g., tables) into which other elements can be inserted, thereby changing their values. Mutable values are shallow-copied. Serialization of static values is straightforward, as there is no aliasing involved. For mutable values, we serialize them according to their reference structure as discussed in §5.4.1.

Data Operations. As mentioned earlier, we include data operations as part of a broader notion of state, in particular as a form of volatile state. Our goal in doing so is to expose them as amenable to a form of "independence" similar to that which we strive to provide for data objects. In particular, if we have fully independent data, then we can propagate changes to the data in terms of descriptions of the operations to perform on the data, rather than the full (and probably mostly unmodified) data itself. For example, when we insert an element into a large set, we can simply propagate "insert element 'foo' into set 'bar' ".

Implementing this independence for operators turned out to be quite difficult. We first identified all atomic operations that can be performed upon Bro script values, for example: binding a value to a global identifier, changing the value of an element in a container, or adding/deleting elements to/from containers. We then implemented a serialization for an abstraction of the operator. The main problem here rests in the need to name a value: when we want to change a value, we need to say which value we mean. Obviously, this is not a problem for static values—they cannot be modified, but only be created and bound to some global identifier (which, by definition, is a name). For mutable values, however, we needed to introduce a naming mechanism, so that one Bro instance can communicate to another which value (e.g., which element of a table) it has modified. We solved this problem by introducing a new, non-user-visible global namespace. Each mutable

value, on which operations are to be tracked, is bound to a hidden identifier unique among multiple instances of Bro (by incorporating hostname and process ID into its name). If multiple instances share independent state, they first agree on these names. Subsequent operations are then expressed in terms of these names.

Events. An event is simply a name plus a set of typed values. Therefore, serializing an event is easy, given that we already have the mechanism for values in place. When de-serializing it, we ensure that the argument types match the prototype of the event handler to avoid inconsistencies leading to run-time errors.

Function Calls. Unlike event handlers, function calls can return values, and hence need to be performed synchronous rather than asynchronous. This difference has major implications for state independence: remote function calls would incur potentially lethal blocking delays waiting for calls to complete and return.

We wondered what the impact of *ignoring* return values of remotely called Bro functions would be. There are two types of Bro functions: internal functions accessible from the script level but defined within Bro's event engine core, and script functions.

We examined all internal functions as of Bro version 0.8a39 (about 100 total), and found only a few that actually have semantics that require both being called remotely and returning a value. It turns out that all of these can either be replaced by some additional script-layer logic, or they are not in fact used by any of Bro's default scripts. These functions all query internal analyzer state. The first group, `active_connection`, `connection_exists`, `lookup_connection`, and `connection_record`, could be replaced by using `active.bro`; for `get_login_state` we could add a new event; `get_orig_seq` and `get_resp_seq` are only used in `terminate_connection` which (usually) has to be called locally anyway; and `get_matcher_stats` and `get_contents_type` are unused.

Considering script functions, we see that instead of calling them remotely, we can as well call them locally, leveraging synchronized user-level data if necessary for their operation as discussed in §5.4.2. Thus, we chose to ignore return values for remote function calls, i.e., we essentially treat them in the same manner as events.

Signatures. A final type of state in Bro, currently static in nature, is the set of byte-stream signatures used by its signature engine (see §6). We would like to convert these to dynamic state, enabling us to change signatures on-the-fly and send them from one Bro instance to another. Implementing this change, however, is quite challenging because the optimized data structures Bro uses internally to match signatures with high performance are not amenable to incremental updates. Changing a single signature currently requires recomputing the entire decision tree used to determine which rules are candidates for matching a given byte stream (prior to regular-expression matching). Because we have not yet tackled making this type of state independent, we do not discuss it further.

5.4.2 Using Independent State

The framework presented in the previous section provides the mechanisms to serialize Bro's state into a portable binary representation. Leveraging this, we now discuss the last step of the transformation from internal state to independent state: we achieve temporal independence by storing state serializations in files; and we achieve spatial independence by propagating state serializations over the network to other Bro instances.

In this section, we present the interface to both types of independent space from the user's point of view (i.e., the person writing policy scripts). From this perspective, the visible aspects of the independent state concept are state files and network connections. Accordingly, the script-layer provides the user with new elements to define which state is to be stored or propagated, respectively. For the user, the complexity of the internal framework remains hidden. Thus, we do not systematically iterate through all kinds of states as we did in the previous section.

We note that the development of the interface has been driven by the needs of particular applications, and will continue to evolve as additional need arises. Moreover, we are in the process of devising an generic event model that is suitable for scalable policy-controlled distributed analysis [KS05].

Temporally Independent State

To make state temporally independent, we make it persistent by storing serializations of the state in files. The state is saved just before a Bro process terminates, and re-read when a new instance starts up.

For user-level data (i.e., script variables) we let the user selectively define which values to save by adding an attribute `&persistent` to the type declarations. For example,

```
global saw_Blaster: set[addr] &persistent;
```

declares a set of addresses for which any changes to the set will be propagated to future invocations of Bro. Such a set is useful, for example, in tracking which addresses have already generated alerts in the past in order to reduce the volume of future alerts (since our policy may be that once we've detected a Blaster infection, we don't need to hear about it again.) Furthermore, because temporally-independent state includes its associated timestamps and timers, we could also use:

```
global saw_Blaster: set[addr] &persistent &create_expire=30days;
```

and Bro will delete each set element 30 days after it was added, so we will be reminded of all still-active Blasters once a month.

Along with `&persistent`, we also added a script function `make_connection_persistent`, which tells Bro to store the event-engine state associated with a particular connection (including related connection timers). There is a corresponding default policy script that uses this function to automatically save state for all connections belonging to a user-definable set of services (e.g., FTP and SSH).

The reason we structure the interface so that the user has to explicitly mark which state to keep persistent, with all other state by default remaining non-persistent, is both

that the volume of the entire set of state can be very large, and also that we find that current policy scripts are often written in a style that presumes that state exists only during the execution of a single instance of Bro. We return to this point when we discuss selective checkpointing in §5.5.1.

In addition to automatically saving state at termination, the new script function `checkpoint` can be called at anytime to store the current set of persistent state (i.e., all data declared `&persistent`, and all connections marked by `make_connection_persistent`) into a state file. Another new default policy script uses this function to save the state at regular time intervals.

The new function `rescan_state` reads such a state file back from disk. By calling this function, we can incorporate a state file generated by another Bro instance during run-time. One application for this is to transfer data between two Bro instances. Another is more interesting: on-the-fly configuration changes. We have added a new command-line option that tells a newly started Bro instance just to write all user-level state into a file, terminating afterwards. By incorporating the resulting file into a running instance, we can change its configuration during run-time—both the values of its global variables and also the definitions of its functions and event handlers, i.e., we can dynamically change the code it executes.

Along with script variables and function definitions, we also developed a way to make event generation temporally-independent. By calling the new function `capture_events`, a policy script can tell Bro to write all events raised during run-time into a file. One possible use for this is to later replay these events in another instance of Bro for debugging or for exploring alternate analyses. This can be very helpful for debugging scripts, as we do not need real network packets to reproduce a situation. Although this is not suitable for all cases—if a script depends on event engine state, replaying events is not sufficient—it works well in many situations.

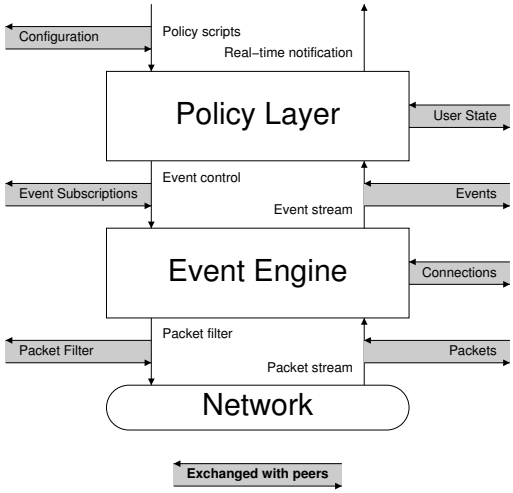
There is another application for temporally independent state: printing it. We can do so either in a "pretty-printed" human-readable form (see §5.5.4), or encoded in XML. We expect the XML output to become highly useful for traffic analysis independent of the task of intrusion detection.[3] For example, when combined with event capturing, it gives us a more abstract view of network activity than raw packets, but remains machine parseable. In addition, when coupled with an XML editor, this facility will enable us to directly manipulate Bro's state. While such "outside the system" editing has the potential for introducing nasty, hard-to-find bugs, they can also prove to be life-savers during emergencies that sometimes arise operationally.

Spatially Independent State

For spatially-independent state, we need to transfer state from one NIDS instance to another one running concurrently. While one way to do this would be via the already-mentioned `checkpoint`/`rescan_state` functions (coupled with manually copying the

[3] At this point, the implementation of XML output is only prototypical; not all internal Bro classes have yet been fully instrumented. Doing so will be straight-forward though.

Figure 5.2 Integrating spatially independent state into Bro's architecture.



file), doing so would be crude and quite limited in power as it would hide the presence of multiple Bro's from one another.

A more direct way is to establish network connections between Bro instances. To do so, one of the instances calls the new function `listen`, which opens a port on the local host waiting for connections from other instances.

```
# Listen on 10.0.0.1:47756 for SSL-authenticated connections.
listen(10.0.0.1, 47756/tcp, T);
```

These in turn initiate connections by means of the new `connect` function:

```
# Connect to 10.0.0.1:47756, using SSL. Retry after 1 minute
# if connection breaks down.
connect(10.0.0.1, 47756/tcp, 1min, T);
```

In the following, we refer to Bro instances which have established a direct connection between them as *peers*. Once a connection is set up, there are several ways for exchanging state. Figure 5.2 shows where we integrated them into Bro's architecture. Using the script-function `request_remote_events` one side can subscribe to a set of events, meaning that whenever the other side generates one of the events, it automatically forwards it to the other side:

```
# Request all HTTP events from peer.
request_remote_events(10.0.0.1, 47756/tcp, /http_.*/);
```

At the receiving end, the event looks just like one generated locally. If required, the script can distinguish between local and remote events by calling the function `is_remote_event`. Additionally, by calling `event_source` it can retrieve more information about the originator.

In addition to sharing events, multiple Bro instances may share data as well. When a script-level variable is declared as `&synchronized`, modifications to its value will be propagated to all peers. For example:

```
global saw_Blaster: set[addr] &synchronized;
```

will cause the script variable `saw_Blaster` to be synchronized with each peer.[4] Any change made by one of them to the set will be transparently reflected in the value of the set as seen by the others.

Furthermore, we can explicitly request the full set of persistent state from another host, reinstantiating it locally. This enables applications such as a dynamic handover between instances (see §5.5.1).

We implemented synchronized tables by propagating changes to the data in terms of operations (see §5.4.1) rather than the full (and probably mostly unmodified) data itself. This can in some circumstances however lead to race conditions. For example, consider a synchronized table that counts alerts generated by a particular source address. If we have parallelized the NIDS processing by having two processes performing different types of analysis, then each of them might determine that the same source address has generated an alert. If the two processes share a spatially independent table, then they might experience a race condition when both processes are modifying the table at the same time; the winner of the race will overwrite the new value provided by the loser. The net effect is that the source address may be charged with only one new alert against it rather than two. Avoiding such race conditions would require mutually-exclusive data operations, for example by using a token-based reservation system [TS02]. However, this would violate Bro's real-time processing constraints: before performing an operation, an instance would have to wait until access is granted. As this is infeasible, we explicitly resort to a model of *loose synchronization* which may exhibit some race-conditions.

Nevertheless, there are ways to mitigate the difficulties caused by race conditions. If in the above example the action being performed is indeed incrementing an alert counter, the operation is in fact not "set it to the value $n + 1$" but rather "increment it". If we propagate this operation rather than the resulting value $(n + 1)$, then the increment will be performed twice and we obtain the correct value in the table of $n + 2$. Accordingly, "increment" and "decrement" are two types of operations which our implementation explicitly supports. Thus, there are no synchronization problems when using Bro's `++`/`--` operators on independent data (with the exception of a potential lag until all instances execute the operation). For operations that we cannot treat in this fashion, we include the old value when propagating an event. For example, the operation "assign 7 to the global `alert_level`" is propagated as "assign 7 to the global `alert_level`, its previous value was 12". This enables us to at least detect and report de-synchronizations. In the example, when changing `alert_level` to 7, we notice that its value before the change is not in fact 12.

[4] Currently, there is no "routing" of state, i.e., state is only exchanged between peers. If more than two Bro instances are to cooperate, they need to be fully meshed. It is part of our ongoing work to devise a more scalable mechanism for sharing spatially independent state [KS05].

To share the event engine's control state, the new function `send_capture_filter` sends a `tcpdump` filter to a peer which then decides (as discussed in the next section) whether to install it. For sharing packets, the function `send_current_packet` propagates the most recently processed network packet. This can, e.g., be used for capturing packets which triggered some activity.

5.4.3 Robust and Secure Communication

Clearly, inter-NIDS communication requires robust and secure operation. Regarding robustness, a key aspect is that, from the perspective of the NIDS process's main functionality, inter-NIDS communication should be unobtrusive. In particular, inevitable networking difficulties such as timeouts or unexpected termination should not perturb the NIDS's operation. Therefore, rather than adding a network communication component directly into Bro's current event engine / script interpreter structure, we instead spawn a second process exclusively dedicated to handling the communication with the peers. The two processes communicate by means of a Unix pipe. (We did not use threads to ensure separate address spaces.) On multi-processor systems, using two processes has the additional advantage of making use of a second CPU. Subsequently, we refer to the two processes as *main process* and *communication process*, respectively.

One key element of our design is to base it on unidirectional messages. This means that while two peers may both send state over the same network connection, Bro's processing never expects one side to reply to something the other side sent. In particular, we do not use any form of application-layer acknowledgments. While this restricts error detection and handling somewhat, it significantly eases implementation as it avoids having to deal with unreceived replies (which would require timeouts and a failure-recovery scheme). We believe that the decrease in complexity wins us more in terms of robustness than we lose in terms of error processing.

The only major drawback of this design decision is that we cannot remotely call functions that return a value. But per the discussion in §5.4.1, the inability to return results is not a severe limitation. Finally, we note that the unidirectionality of communication only affects the core-level communication between two instances. For example, it is still possible to build a script-level handshake mechanism by passing a sequence of events between two peers. In fact, the handover mechanism shown in §5.5.1 does exactly this.

In §5.4.2 we discuss how the NIDS's real-time constraints leads us to abide loose synchronization semantics, i.e., the possibility of race conditions. For similar reasons, our communication design does not make any timing guarantees. For example, transferring large amounts of data may delay the reception of an event. Also, while all state from one endpoint always arrives in the order in which it was sent, state from multiple endpoints may be received intermixed.

Along with designing for robust communication, we also need to secure the communication, i.e., providing for confidentiality and authentication. We do so via SSL (implemented using OpenSSL [Ope]), configuring peers with the public keys of trusted certification authorities (CA).[5] Enabling encryption for confidentiality is straightforward.

[5] We preferred SSL over other means, such as IPSec, because of its ease of use: while we have to (slightly)

For authentication we use signed certificates. A peer only approves a connection if the other endpoint presents a certificate signed by one of the CAs. We note that we always have both peers authenticate themselves (in contrast to server-only SSL authentication as it is often used).

Finally, a NIDS necessarily gathers a great deal of information, the access to which is often restricted and not to be shared with other parties. Therefore, trust is a vital issue when allowing a peer to connect to a Bro instance. Given the new capabilities presented above, without further restrictions any peer could access the instance's state as well as send it arbitrary, perhaps misleading, state. In general, for a Bro instance we can identify four levels of increasing trust in a peer:

1. A peer may not talk to it at all.

2. A peer may get state from it, but it does not accept any state.

3. A peer may get state from it, and it accepts non-control state.

4. A peer may get state from it, and it accepts all state (including control state such as sending us a filter that enables more analyzers).

The need for these levels immediately arises operationally. For peers with which we have not developed any monitoring agreement, the first level of trust (no access) is appropriate. For sites that we wish to help but we do not know if they themselves are run competently, the second level is prudent. For sites with which we have a close working relationship, the third may be. Finally, when using multiple NIDS instances internally, for example for load-balancing, the fourth level likely makes sense.

Accepting state from a peer (i.e., trust level 2 and above) illustrates another implication: due to the complex semantics of the NIDS' state, we cannot really validate that the input is well-formed. Consider for example shallow-copied objects: when a serialization references another object, it only includes an unique ID. Upon de-serialization, there is no apparent way to validate that the referenced object is semantically correct (we do ensure type safety though). Thus, accepting state entails trusting the peer to send valid data, and this must be kept in mind when assigning trust levels.

All four levels rely on a correct identification of the remote side, which we enable via SSL-based authentication (and firewalling to enforce the first level). While we have not currently implemented an explicit framework to directly state "peer x is on trust level y", we provide hooks to enforce such trust-levels at the script-level. For example, when a remote peer has successfully connected, the event engine generates an event to tell us so. Our policy script can then decide whether we want to accept state from the peer. If we receive control-state (which currently can only be a packet filter), the decision to install it is again left to an appropriate event handler.

adapt the NIDS, the user does not need to install additional infrastructure. This is particularly important for fostering distributed, independently-administered confederations.

5.4.4 Integrating External State

While we performed our initial implementation of independent state in the context of the Bro NIDS, the concept applies more generally to other applications as well. In principle, *any* system, not just other NIDS instances, may chose to make its fine-grained internal state accessible to peers.

As a major step in this direction, a lightweight, highly portable library called *Broccoli*[6] [Brob] has been developed at the University of Cambridge, UK, which enables arbitrary applications to partake in the exchange of Bro's state. Broccoli is an independent implementation of (parts of) the serialization/communication protocol that we have developed for the Bro system. Broccoli nodes can request, send, and receive events just like Bro instances can. In §5.7, we demonstrate the power of such NIDS-external independent state by supplying the Bro system with host-based application context.

5.5 Applications

We now describe several powerful applications of independent state in network intrusion detection. We first show how we can use independent state to greatly enhance Bro's traditional model of regular checkpointing, including support for robust crash recovery. Then we discuss distributed intrusion detection, concentrating on the utility of spatially independent state. Finally, we show how independent state can be used for dynamic reconfiguration, profiling, and debugging.

We easily implemented each of these applications, combining independent state with Bro's flexibility. Although at first blush each might seem to be yet-another-extension of Bro's generally-extensible functionality, the ease of implementation highlights the power of the approach.

5.5.1 Selective Checkpointing

As we see in §4, NIDSs face fundamental state management problems. Bro's original approach for large-scale state reclamation used to be the brute force approach of simply starting over from scratch. That is, to run Bro 24x7 we (and other Bro users) resorted to checkpointing (see §2.4.4). The main advantages of checkpointing are its simplicity and the robustness it provides. But, clearly, simply throwing away all of a NIDS's state at certain times is not the best approach. While with the work presented in §4.5, checkpointing can now be replaced with more a fine-grained dynamic state management, occasional restarts are certainly still necessary. In both cases—traditional checkpointing and otherwise required restarts—we want to retain a selected subset of important state, while reclaiming all of the rest.

For Bro, the two main types of state lost when checkpointing are internal connection state (including analyzer-specific state and attached timers) and script-level data. However, the concept of persistence described in §5.4.2 enables us to choose individual connections (by calling `make_connection_persistence`) and specific script-level data

[6] *Broccoli* is the healthy acronym for "Bro Client Communications Library."

(via `&persistent` declarations) to transform into independent state, thus enabling a new Bro instance to use them as part of its initial state. Doing so allows us to perform longer-running forms of analysis uninterrupted, such as tracking slow scans, long-lived interactive connections, usernames, inferred software versions, alerts already generated, and addresses that Bro has blocked in the past using its dynamic blocking facility.

While temporally-independent state thus enables us to keep key state across restarts, implementing it soundly also requires a dynamic *handover* mechanism. The problem here is that the currently executing instance of Bro has to save its persistent state at some specific point in time, *after* which the new instance can begin executing. If we have to wait for the new instance to start up, we will incur a monitoring outage.[7] We solve this problem by using spatially independence instead of temporally independence. We implement dynamic handover by starting up the new instance and having it connect via a (local) network connection to the old instance, requesting its current set of persistent state. After this has been successfully transmitted, the old instance terminates itself, and the new one starts processing.

As already discussed, we do not simply make *all* state persistent. Doing so would defeat the purpose of checkpointing. Therefore, the next step is to identify the state for which this makes sense. For our operational environments, we keep internal connection state for interactive services that tend to have long-lived connections such as FTP and SSH. (This list is easily customizable. In addition, one could choose to add connections for which some malicious activity has already been seen.) For script-level data, we took Bro's default policy scripts (as of version 0.8a57) as representative for the usage of state in Bro scripts. Our first observation is that nearly all of the scripts store their relevant data in tables or sets. We found five basic usages:

1. Remembering messages already logged to avoid duplication.
2. Remembering hosts which have done "something" (e.g., propagating a worm).
3. Remembering additional state for connections (e.g., which FTP data connections have been negotiated by a control channel).
4. Holding configuration data, such as particular hosts allowed to do "something". (e.g., connect to a certain host; this data is more or less fixed).
5. Remembering additional data derived from the script's analysis (e.g., software installed on a host).

Taking the MWN environment as a test case, we made all tables belonging to the first group persistent. Most of these tables are rather small in volume[8], and suppressing unnecessary log messages is a vital NIDS capability [Axe99b, Jul03]. For the second group, we differentiate between short-term (minutes or less) and longer-term data. The former is often quite large in volume and often not worth keeping. For example, the

[7] If we start the new instance first, and have it read in the persistent state while already processing packets, we incur significant analysis inconsistencies.

[8] With the notable exception of the table `weird_ignore` recording all the "crud" (see §3.3.3).

script recognizing the Blaster worm [Bla] by its scanning activity keeps two tables: one tracking pairs of hosts that have communicated over TCP port 135 within the last five minutes, and the second remembering all already-identified worm sources. We decided to make only the latter persistent.

The third group (remembering additional state) is more problematic. Ideally, we would like to keep information related to any persistent connection, but discard all the rest. But to do so the scripts would need significant restructuring, as their semantics vary too much to automatically deduce which information is associated with persistent connections. There are some tables, though, for which we know they always correspond to state for persistent connections. For example, the FTP analyzer script remembers the FTP connections. We made these kinds of tables persistent, but left all other tables unmodified (i.e., ephemeral).

We also left the fourth group untouched, as configurations are mostly static and better changed manually if the need arises. Finally, for the last group we found we needed to make case-by-case decisions. For example, to derive vulnerability profiles (see §6.2.4) one of the scripts detects the software used by hosts, an excellent example of information we declare `&persistent` so we do not lose it.

We observed that making a table persistent almost always implies adding an user-level timeout (see §4.5.2), too, as it generally does not make sense to store state forever. One exception we made, however, was for vulnerability profiles, where we prefer to keep the information as long as possible. If required, we can always delete it manually.

5.5.2 Crash Recovery

A related application of independent state is better crash recovery. The three main reasons for the crash of a NIDS are resource exhaustion, attacks, and programming errors [Pax99]. In most systems, including Bro, with a crash we lose all the state so far collected by the system. By using the `checkpoint` function (see §5.4.2) regularly, however, we can significantly mitigate the effects of crashes, so that we only lose data accumulated since the last checkpoint.

Our experience is that crash recovery is invaluable. This is not only the case when actively developing the IDS—where we often experience crashes due to programming errors, and, hitherto, always loose the complete state of system as a consequence— but also in a production environment, where crashes are still a fact of life, particularly due to resource exhaustion. Not only does crash recovery allow us to continue operating with only a minor loss of state (in terms of the importance of the state), but the checkpoint also allows us to analyze the state shortly before the crash post mortem (see §5.5.4).

5.5.3 Distributed Analysis

Now that we have a means for a NIDS to communicate its state, we can use that mechanism to distribute its analysis. To date, distributed NIDS have generally imposed a specific model on the form of distribution, e.g., the sensor model (see §2.3.3). Independent, fine-grained state opens up new degrees of flexibility for distributed analysis. In

this subsection we look at three different approaches, all of which we have been able to implement and experiment with by having added independent state to Bro. The first approach supports load-balancing for monitoring high-volume links. The second supports the sensor model. The third looks at propagating information between otherwise decoupled systems.

Load-Balancing

As discussed in §4.4, it is exceedingly difficult to analyze a high-volume packet stream with a single NIDS. One strategy for coping with such a load is to distribute the analysis across several machines, each responsible for some part of the work. A key question then is how to coordinate their operation. Currently, using Bro operationally we do this by running several independent instances on different slices of the network traffic. But without any state sharing, this loses important information. Our goal is to retain the depth of analysis a single Bro could in principle achieve if it could cope with the load.

To this end, we first need to decide how to divide the traffic between the multiple systems. We can either do so statically (each system gets all packets matching some fixed criteria) or dynamically (e.g., for each connection we decide individually which system will analyze it). Our initial efforts have focused on static approaches due to their simplicity, distributing based on *(i)* the local IP space, or *(ii)* application protocol.

Dividing by IP Space. Fruitfully splitting up the local IP space requires knowledge of the network to find a division such that individual NIDS instances receive comparable loads. From our operational experience, measuring the volume and leveraging the expertise of the network's administrators to do so is not hard. The main advantage of distribution based on dividing the IP space is the ease of further distributing the load by introducing additional systems. The main disadvantage is that, without any communication, we cannot correlate activity between different subnets anymore, such as detecting scans.

To assess the feasibility of this approach, we examined the Bro 0.8a53 policy scripts to determine the degree of communication they require. We found that there is one dominant case where without communication we lose information: several scripts store information about individual hosts, usually of the form "host a.b.c.d did something [n times]". For example, the worm detection script keeps a table storing all already-known worm infectees. Not propagating this state among the concurrent Bro's would have two effects: *(i)* each of the instances would alert individually if it recognizes the worm, and *(ii)* more importantly, if an instance has not yet identified the worm by itself, it obviously cannot use this information in other contexts (e.g., treat signatures matching a known worm infectee different from other matches).

With spatially independent state, however, we can easily solve these problems by declaring the tables `&synchronized` (per §5.4.2). Now each instance propagates its state to the peers. In fact, this is an ideal situation for loose synchronization (see §5.4.2); the propagated updates are "add this element to the table" and "increase

this element's counter". Both are independent of the order in which they are applied. [9] Also, for this task, a short interval of de-synchronization does not hurt.

Dividing by Application. To divide the load by application, we delegate applications that make up a significant share of the load to dedicated systems. If, for example, there is a large fraction of HTTP traffic, we could exclude HTTP processing from the main system and move its analysis to another machine. This is in fact what is done operationally at LBNL. But this approach lacks general scalability: the load is only significantly reduced if the NIDS does indeed spend quite some time processing the particular application. For Bro, this is true for HTTP (due to Bro's detailed analysis of the HTTP sessions), and also for a few other applications, but these total only a handful. Thus, the scalability of this approach is limited.

Again we examined the scripts to assess where division by application would require inter-Bro communication. In the most cases, no communication is needed for application-specific analysis. However, there a few exceptions. One is the FTP analyzer. It parses the PORT negotiation of FTP data connections to classify the subsequent corresponding connections as such if it sees them. Another problem concerns analyzers that need to see traffic from all applications, such as Bro's scan detector. To detect vertical port scans, it counts connection attempts to different ports (applications) per host. Other examples include the ICMP analyzer correlating ICMP 'unreachable" messages with corresponding connections, and the analyzer that derives vulnerability profiles (see §6.2.4).

We note that the two techniques of dividing by IP space and dividing by application can in principle be combined (dividing by both) in environments with a large number of analysis hosts available.

Sensor Model

A well-established architecture for distributed network intrusion detection is the sensor model (see §2.3.3). Bro is conceptually well-suited for this kind of deployment. Its architecture already clearly separates between low-level and high-level analysis by means of its division into event engine and policy script interpreter. The main interface between these two layers are the events. So, the most obvious way to apply the sensor model to Bro is to spatially separate the event engine from the script layer (i.e., run them as separate processes). This is easy to achieve using spatially independent state.

However, as we discuss in §5.6, propagating large volumes of state comes at a non-negligible cost. Therefore, while propagating all of the event layer's information will work well for smaller setups, it does not scale to high-volume networks. Hence, in such environments it is more promising to partition the processing a layer up. That is, the sensors perform the usual script-level analysis in addition to their event engine processing, with those scripts synchronized as discussed above. Then we dedicate an

[9] As long as elements are not removed. But in most cases this happens, if at all, due to automatic expiration, which each instance does by itself.

additional Bro instance to correlating their combined output at a meta-level, for example by using any of the methods discussed in §2.3.5.

As a first step in this direction we implemented a simple but operationally very fruitful extension: combining all log messages coherently in a single place. In the MWN setup, where the load is divided among multiple Bro instances, a central server now receives all output in real-time. We note that while this is available in other distributed NIDSs, their communication is often *restricted* to the exchange of log-like messages. With our architecture, centralized logging is an almost trivial application: each script-level log statement generates an event which is propagated to the central server.

Propagating Information

Another potentially valuable application of spatially independent state is using it to tell other systems results of our analysis. We discuss two examples here: intensifying analysis of suspicious hosts and propagating IPs that a NIDS has chosen to dynamically block.

Suspicious Hosts. As mentioned above, due to the large load on a high-volume link, a single system cannot perform a detailed analysis on the full traffic. One solution is to run only coarse-grained analysis on all of the traffic, but to intensify the inspection for hosts found to be conspicuous. For example, often administrators observe that attackers first perform scans of the network before actually targeting some hosts. Large scans are easily detectable using coarse-grained analysis. After identifying a scanner, we can then look at the packets coming from the same source in more detail.

We implemented this by running two instances of Bro. The first instance watches a large fraction of the traffic but only runs a modest set of policy scripts (most notably the scan detector). When it generates an alert for some host, it also sends an event containing the host's IP address to the second Bro instance. By default, the second instance does not see any traffic at all. Yet, if it receives such a suspicious address, it modifies its analysis filter to include all packets coming from that source. In addition to using more scripts and a large set of signatures, it records the complete set of packets to disk.

Propagating Blocked Hosts. The LBNL environment currently runs several Bro's at different entry points into the network. As discussed in §3.3.1, LBNL's security policy includes dynamically blocking scanners detected by Bro by modifying the entry router's access control list. Because not infrequently a scanner first probes a set of addresses corresponding to one entry point and then later another set corresponding to a different entry point, there is considerable operational interest in enabling the different Bro's to communicate their blocking decisions to one another. We plan to support this by modifying the operational Bro's to broadcast events to their peers when blocking new hosts. We also plan to experiment with broader sharing of such information across different institutions (e.g., MWN, LBNL, and UCB)

5.5.4 Dynamic Reconfiguration, Profiling and Debugging

A final set of applications for independent state leverages the broader notion of independent state encompassing not only data values but also functions and policy script event handlers. Here, we realize the benefits made possible by converting Bro's static state into dynamic state. Such independent state allows us, first, to both tune and retarget the system without having to restart it; and, second, to inspect the system's state during run-time in several different ways.

Dynamic Reconfiguration

We can use the independence of broader forms of state (functions and event handlers) to dynamically reconfigure a running Bro. This provides operational flexibility and supports tuning. In terms of operational flexibility, frequently during daily operations the need arises to change the configuration of the NIDS in response to a newly perceived threat or problem. For example, we have detected a break-in and now want to alert on any return of the attacker; or, we have learned of a new attack pattern and want to immediately start using it; or, a new source of benign traffic has appeared which is overwhelming the NIDS and we want to skip processing it for now. These can all occur in a "fire-fighting" mode, i.e., we really need to deploy the change immediately. With independent state, we can introduce such changes—including modified function and event handler definitions—directly, without incurring the loss of fine-grained state that would occur even when using our enhanced checkpointing.

Another use of dynamic reconfiguration is to support tuning, i.e., optimizing the NIDS's configuration for the local environment. From our experience, one of the most common problems with making configuration changes for tuning is that the effects of the changes often do not show up immediately. Many changes become only visible when the system has built up a significant amount of state, which can take a long time after a conventional restart. This is particularly true for configuration parameters like timeouts and thresholds. We can ameliorate this problem by collecting traffic traces and testing against them off-line, but such traces can be huge and unwieldy to work with.

While our enhanced checkpointing can help, it does not fully solve the problem. Often when making many small changes in a short time, we do not actually want the controlled loss of state which checkpointing achieves, but prefer to keep *all* state. We want ideally to have the system just pick up the changes and keep running, similar to the fire-fighting changes discussed above.

The way we do such on-the-fly changes in practice is as follows. Consider an already-running Bro whose configuration we would like to change in some respect. We first make the modification to the configuration, i.e., edit the scripts. We then convert the full configuration into a state file using the new command line option described in §5.4.2. Finally, we copy the file into a directory regularly checked by the running instance, which notices the update, loads it, and switches to using it. None of the other state is lost. (We provide a new policy script to activate the regular check of the directory.)

Figure 5.3 Visualizing state of the scan detector (addresses anonymized).

```
ID reported_port_scans = {
 [10.11.184.36, 10.126.197.84, 100]  @01/21-12:11
 [10.112.68.194, 10.45.144.114, 1000]  @01/21-11:00
 [10.112.68.194, 10.45.144.114, 100]  @01/21-10:59
 [10.112.68.194, 10.45.144.114, 10000]  @01/21-11:01
 [10.112.68.194, 10.45.144.114, 50]  @01/21-10:59
 [10.11.184.36, 10.126.197.84, 50]  @01/21-12:11
}
```

(a) Subset of reported port scans at time T.

```
ID reported_port_scans = {
 [10.112.68.194, 10.45.144.114, 50]  @01/21-10:59
 [10.112.68.194, 10.45.144.114, 10000]  @01/21-11:01
 [10.112.68.194, 10.45.144.114, 1000]  @01/21-11:00
 [10.184.146.140, 10.74.170.189, 50]  @01/21-13:16
 [10.11.184.36, 10.126.197.84, 100]  @01/21-12:11
 [10.11.184.36, 10.126.197.84, 50]  @01/21-12:11
 [10.112.68.194, 10.45.144.114, 100]  @01/21-10:59
 [10.184.146.140, 10.74.170.189, 100]  @01/21-13:16
}
```

(b) Subset of reported port scans at time T + 1.5 hr.

```
  [10.112.68.194, 10.45.144.114, 1000]  @01/21-11:00
  [10.112.68.194, 10.45.144.114, 100]  @01/21-10:59
  [10.112.68.194, 10.45.144.114, 50]  @01/21-10:59
+ [10.184.146.140, 10.74.170.189, 100]  @01/21-13:16
+ [10.184.146.140, 10.74.170.189, 50]  @01/21-13:16
  [10.11.184.36, 10.126.197.84, 100]  @01/21-12:11
  [10.11.184.36, 10.126.197.84, 50]  @01/21-12:11
```

(c) `diff` of (a) and (b).

Profiling and Debugging

Another significant problem when operating a NIDS is understanding its behavior during run-time. When developing policy scripts, we find they can work in unexpected ways, due to either programming errors, or to encountering network traffic with different characteristics than expected. These kinds of problems are very hard to track down, as often they only manifest themselves after a considerable amount of time. This means that they cannot be reasonably targeted with Bro's existing tracing and interactive debugging mechanisms.

We find it is a great help if we can take a look at Bro's current state. With independent state, this is easy to achieve, since the files generated by `checkpoint` contain all the necessary information and can be converted into a human-readable ASCII representation.

Figure 5.4 Visualizing access patterns of a script function (from `scan.bro`).

```
ID check_hot = check_hot
(@01/21-12:23 #4715580)
{
local id = c$id;
local service = id$resp_p;
if (service in allow_services ||
    c$service == "ftp-data")
  (@01/21-12:23 #2932175)
  return (F);

if (state == CONN_ATTEMPTED)
  (@01/21-12:23 #1138955)
  check_spoof(c);
[...]
}
```

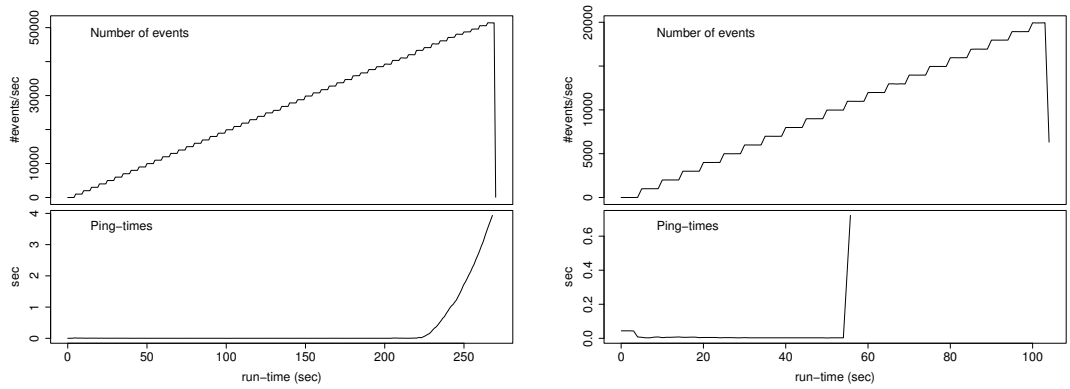
For example, in Figure 5.3(a) we see the table containing all known port scanners at some point of time. The output includes timestamps when the entries were last accessed. In Figure 5.3(b), we see the same table about 1.5 hours later. We see that two entries have been added. For larger tables, the differences may be hard to see, but the ASCII output formats are suitable for processing with Unix utilities such as `sort` and `diff`, as illustrated in Figure 5.3(c). Now the differences are obvious; we can *see* the scan detector working.

Along with data values, our implementation of independent state provides timestamping for script functions as well. Figure 5.4 shows a checkpoint of the `check_hot` function from the default scan detection script. The different basic blocks in the code are annotated with timestamps indicating the last time they were executed, and with counts of how many times they have been executed. These annotations are extremely valuable for profiling, assessing code coverage, and detecting stale script elements. We can again use tools like `diff` to dynamically track which portions of the code are being executed, and how frequently. The counts are particularly useful: if they differ significantly from what we expect, there is either a coding error, or a misunderstanding of the network traffic. Finally, we aim to eventually use such information for long-term policy script maintenance, for example by locating policy script elements that have become stale and are no longer needed.

5.6 Performance Evaluation

To examine the performance of our architecture in more detail, we concentrated on the communication system, as this encompasses most of the other components (e.g., the serialization framework). First, we used synthetic stress-tests to gauge the maximal

Figure 5.5 Propagating increasing number of events.



(a) Plain connection.

(b) SSL connection.

throughput. Then we turned to real-world traffic captured in the MWN environment to assess the system's performance on realistic data.

For all of our experiments, we used two machines, one acting as sender and one as receiver. Most of the measurements were performed on MWN monitors 1 and 2 (see §3.2.1). The experiments involving live-capture of network traffic were done on MWN monitors 3 and 4.

5.6.1 Benchmarks

First, we instrumented the sender to emit events at configurable rates. Starting with no output at all, we increased the rate by 1,000 events every 5 seconds until either the sender or the receiver could not handle the load anymore. Simultaneously, the sender sent out *ping* messages every second to which the receiver responded with *pongs*. We refer to the duration of the lapsed interval until the pong returns as a *ping-time*. At the time the sender's main process sees the pong, the ping/pong combo has traversed four different processes (sender's main, sender's communication, receiver's communication, receiver's main, and back). Thus, the ping-time is a measure for the lag which the communication introduces. The smaller the ping-times, the faster events (and other information) are propagated. If the ping-times start to increase, it is a sign that some queue on the path is filling up, i.e., a limit has been reached.

Figure 5.5(a) shows the rate of emitted events as well as the ping-times. We see that with increasing output the ping-times were roughly constant (with a median of 2 ms) until the sender's rate reached about 44,000 events per second (which comprise a volume of 8.3 MB per second). At this point, the ping-times exceeded 0.1 s for the first time and continued to increase. It turned out that it was the receiver that became overloaded. For every received event, its main process had to call a (empty) script-level handler,

which became too much of a burden eventually. In our synthetic benchmark, the sender itself did not raise the events; it only sent them out. Therefore, event handling was not a problem on its side.

Events have different types of arguments. The events used for Figure 5.5(a) carried two parameters: a string and an instance of a compound connection type. The latter is rather complex; it contains others compound objects itself.[10] If we reduce the complexity of the arguments we are able to achieve higher rates, up to more than 100,000 events per second when sending events without any parameter at all.

We conducted similar benchmarks with state-propagating operations (as triggered by `&synchronzized` script variables; see §5.4.2) and full network packets. The results were similar as with events. For state operations, we sent table assignments of the form `t[index]="string"`. We could send up to 58,000 such operations before the ping-times exceeded 0.1 s. Again it was the receiving main process which was not able to keep up. To measure sending raw packets, we transfered a trace captured in the MWN environment. The ping-times began exceeding 0.1 s when the transmission rate hit 16,000 packets per second. This data rate corresponds to more than 11 MB/s, i.e., the volume approached the maximum bandwidth of the 100 Mbps link between the systems.

We repeated the benchmarks with SSL encrypted sessions, finding that the amount of objects that could be transmitted decreased noticeably. Figure 5.5(b) shows a fall-off to 11,000 events per second when the ping-times crossed the 0.1 s limit. Interestingly, it seemed to be the *receiver's* communication process that was the bottleneck. Being busy with decryption, it was unable to accept new messages for tens of seconds. During that period the queue of sender's communication process filled up and it shut the connection down. While we have not yet further investigated this effect, some tuning of OpenSSL could possibly increase the limit.

To summarize, our architecture can transfer tens of thousands of objects per second, and, therefore, seems suitable for use in high-performance environments. However, we note that these benchmarks represent a best-case: the Bro system is concentrating solely on communication. Thus, we now evaluate performance with under packet load.

5.6.2 Performance on Realistic Data

The benchmarks presented in §5.6.1 suggest that the communication framework is not going to be a bottleneck. However, in a high-volume environment, the packet processing itself is already a very demanding task. Thus, we now examine how well the communication blends in.

[10] Such arguments are quite typical for Bro's events. Often more than one event is raised per connection, and they all share the same connection as argument. In fact, if the content of the connection object has not changed, the serializer exchanges only an identifier for subsequent events rather than the full object. In our benchmark, we ignored this optimization and always sent the full object. We note that this represents the *worst-case*.

Pseudo-Realtime

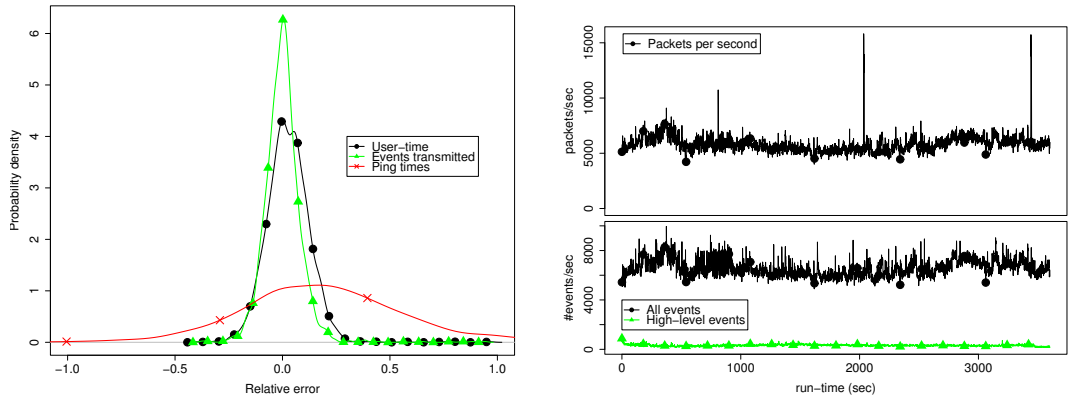
First, we need a methodology to perform *reproducible* measurements. A common approach to evaluate the performance of a NIDS is to capture a packet trace and run the NIDS offline on it with different configurations. However, to evaluate communication performance, this approach does not work well: the NIDS can process a trace more quickly than the corresponding realtime. This leads to its analysis time being "compressed" (we term this *trace time*). Yet, the communication, located in a separate process, is decoupled from the trace time; it is performed in *realtime*.

Nevertheless, we wanted to keep the trace-based evaluation approach for its reproducibility. Hence, we needed to synchronize network time and realtime. To this end, we added a *pseudo-realtime* mode to Bro. If activated, the main process reads packet input from a trace but deliberately inserts delays into its processing. These delays resemble the inter-packet gaps observed when capturing the trace. That is, the processing of a new packet is deferred until the corresponding amount of realtime has passed.

Given the same input, a pseudo-realtime Bro performs the same operations as a live Bro, i.e., the two do not differ in terms of detection. However, the times at which operations are performed could slightly vary, leading to different system and network loads. To ensure that the pseudo-realtime mode does indeed provide similar results in terms of load as running on live traffic, we performed an experiment. We started a Bro process on live MWN input and simultaneously captured the traffic on the same machine, using the same BPF filter as the Bro process. Then we re-ran Bro offline on the trace in pseudo-realtime. We configured both Bro runs to use the default set of analyzers (with some reduced time-outs and including UDP packets). To avoid losing packets, we excluded two high-volume networks from the analysis. The resulting 30-minute trace had a volume of 832 MB, consisting of 4.9 M packets (61.0% TCP, 39.0% UDP). Based on ports, HTTP and DNS were the most prevalent protocols (29.1% and 8.1% of the packets, respectively). The filtered trace contained 1.4 M connections/flows. (We note that these figures are *not* representative for MWN's network traffic but are influenced by Bro's filtering.)

During the runs we logged the user-level CPU utilization of the main process and the number of transmitted events, both per one-second intervals. Additionally, we measured ping-times once per second. Then we calculated the relative errors of the pseudo-realtime figures compared to the live figures. Figure 5.6(a) shows the corresponding densities. For user-time and number of transmitted events, we see a nearly perfect match. The errors were larger for the ping-times. The median of the relative errors was 0.16, the standard deviation was 0.47. That is, in the live run, on average the pings needed longer than in pseudo-realtime. However, the median of the *absolute* errors was 2 ms (standard deviation 5 ms), i.e., the differences were in fact rather small in absolute terms. We assume that these discrepancies stem from the increased system load when running live. In fact, examining the *system* times of the main processes, the median of the relative errors was much larger than for the user times (0.64 vs. 0.02). This is due to the live packet capturing and filtering which takes place inside the kernel.

Figure 5.6 Evaluation of pseudo-realtime.



(a) Bro running live vs. pseudo-realtime.

(b) Number of (filtered) packets and events on trace.

All in all, we believe that pseudo-realtime provides us with reproducible yet realistic measurements. Therefore, we used it to examine the communication framework in more detail.

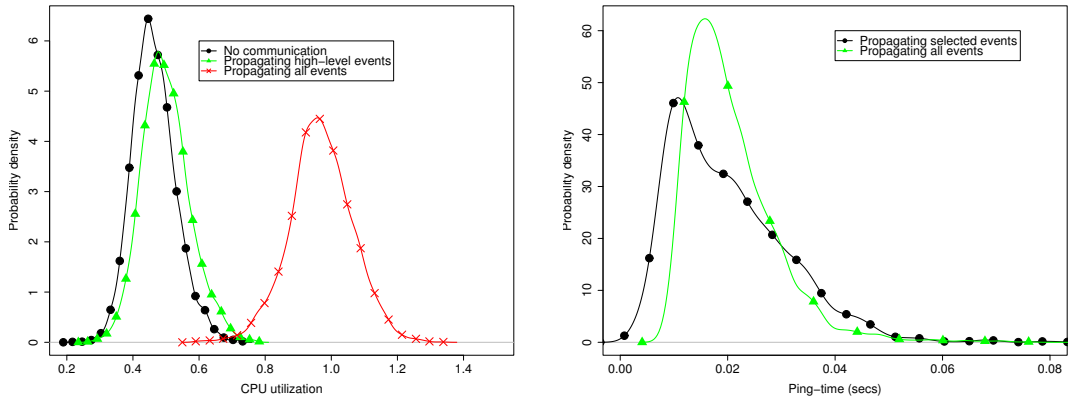
Measurements

Having the pseudo-realtime mode in place, we examined three different Bro configurations to understand the impact of communication. We captured an one-hour packet trace in the MWN environment. We included *all* packets except those of one high-volume subnet. To ensure that we do not lose packets, we captured the traffic with a high-performance Endace DAG card [End]. The trace spanned 1 hour with a volume of 88 GB and a mean rate of 40 K pps. It consisted of 5.2 M flows, 144 M packets, 92.2% TCP and 6.7% UDP, with HTTP and SSH the most common protocols (50% and 6% respectively).

We first run Bro without any communication at all (using the same setup as before). Then we configured it to propagate *all* events to its peer. Finally, we changed the configuration to emit only a subset of events, which consisted of all but the connection setup and tear-down events (and their UDP equivalents raised for basic UDP requests and replies). These events are rather low-level but comprise 95% of all events in our trace. Figure 5.6(b) shows the differences in the number of events over time.

For the three different configurations, in Figure 5.7(a) we see the densities of CPU utilization per second. The measured utilizations are the total sum of user and system time of both processes, main and communication (which is why on the two-CPU machine the values can exceed 1). We see that when emitting all events the average utilization

Figure 5.7 Running different configurations using pseudo-realtime.



(a) Total CPU utilization.

(b) Ping-times.

increased noticeably: the median shifts from 0.46 without any communication to 0.97. Yet, with the subset of events the performance impact is only marginal (median 0.49). Looking at the ping times (Figure 5.7(b)), we hardly see any difference between the two runs which involve communication: with all events, the median of the ping-times was 19 ms while with the subset it was 18 ms. Thus, the communication system performed well within its capacity limits. In general, the ping times are pleasantly low, considering the ping/pong path across four processes which also had to handle a significant packet and event load.

Based on these results we draw two main conclusions. First, our architecture works well enough to support propagating thousands of events even when having to handle a high packet load. However, in such situations it does incur a noticeable overhead: the increased CPU utilization is non-negligible. This implies that simply forwarding *all* events will not scale very well in large installations. Given the amount of traffic, this is hardly surprising. However, we also see that with smaller numbers of events, the performance overhead is not significant at all. Thus, distribution schemes which focus on propagating higher-level events appear to be a very promising approach for large-scale installations.

5.7 Inclusion of Host-Based State

As discussed in §2.3.1, traditional NIDSs are either network- or host-based. However, no approach, taken exclusively, provides a satisfactory solution: network-based systems are prone to evasion, while host-based solutions suffer from scalability and maintenance problems. We now present an approach to leverage the best of both worlds using independent state.

As observed in §5.4.4, independent state is not restricted to being shared between NIDSs. Generally, *any* device can participate in the exchange, providing access to its internal state by means of the same mechanism. We use this approach to integrate host-based components into the NIDS's analysis, allowing them to send and receive state. Thereby, we preserve the advantages of network-based detection, but alleviate some of its weaknesses by improving the accuracy of the traffic analysis with the inclusion of specific host-based context.

In this application of independent state, we focus our attention on crucial and frequently exploited host services that typically run on only a handful of machines. Compared to the usual host-based paradigm of performing all analysis on the end host itself, our solution incurs very modest performance and maintenance overhead on the end hosts because the actual analysis work is performed not by it but on a different system. From the perspective of the NIDS, our approach trades off an additional burden of communicating with the end systems for potentially saving a considerable number of cycles in the analysis process by obviating the need for costly NIDS processing to resolve ambiguity. A key question for the approach is to what degree this trade-off of increased communication for decreased processing is a net gain. As we show, it is indeed a significant win.

We note that the idea of leveraging host-based context in network-based IDSs is not itself novel [AL01, WH01]. The contributions of our work are twofold: first, we move the idea forward by tightly integrating it with the well-established policy-driven approach of the Bro system. Second, we identify novel ways of leveraging the context provided by similar processing stages in the NIDS and host-based applications. In a detailed case study, we instrument the Apache Web server with an interface to Bro. To demonstrate the feasibility of the architecture, we deploy such a setup in two production environments. Additionally, we examine the effectiveness of our multi-point analysis approach in a testbed by launching a large number of scripted attacks against the Web server.

In the following, we first discuss the benefits of including host-supplied context. Next, we present our integration of host-based state into Bro's independent state architecture. Then we conduct a case study: we instrument the Apache Web server to supply information to concurrently executing Bros, and present our experiences with instrumented Apaches in a testbed installation as well as in two productional environments.

5.7.1 Benefits

Having a distributed NIDS at hand, we can use the NIDS's communication mechanisms to implement host-based sensors to supplement the NIDS's analyses. Our motivation for augmenting network-based analysis with host-supplied context is fivefold:

Overcoming Encryption. One major benefit of host-supplied context is that the host has access to information before and after any flow encryption takes place. The recipient of an encrypted connection can be instrumented to report selected information to the NIDS, such as user login names or requested objects. Thus, instrumenting server applications that employ encrypted communication allows us to do the same protocol analysis as for clear-text protocols.

Comprehensive Protocol Analysis. Having host applications report to the NIDS enables us to access additional information about the applications' internal protocol state. As endpoints fully decode the application-layer protocol in any case, they can easily provide the NIDS with context that for the NIDS is hard to derive itself.

A simple example is user authentication during a Telnet login session. The Telnet protocol does not include any information about login success or failure, so Bro must resort to heuristics in an attempt to infer the result of an authentication attempt based on the keystroke/response dialog [Pax99]. But the Telnet server end host knows the outcome of such attempts immediately and unambiguously.

Anti-Evasion. Evasion attacks are one of the most fundamental problems of network intrusion detection (see §2.4.3). They exploit ambiguities inherently present in observing network traffic from a location other than one of the endpoints. These ambiguities render it hard, or even impossible, for a NIDS to correctly interpret skillfully crafted packet sequences in the same fashion as the end host receiving them. However, the NIDS's analysis can leverage host-based context at multiple levels. One way to use this is for learning how the application interprets the received data, i.e., we can use additional information to detect evasion attacks against the NIDS. By including application-layer state of the host into the analysis, such attacks can be detected and/or avoided. Another interesting approach is the instrumentation of a host's network stack, which would allow it to share information about its stream reassembly with the NIDS. A key question here is how to minimize the amount of information that needs to be shared to allow such a comparison. For example, we can envision exchanging checksums of the stream to detect mismatches in a lightweight fashion. Such instrumentation would allow us to monitor *multiple types* of applications for evasion attacks without the need to instrument each application individually.

Adaptive Scrutiny. Generally, there is a wealth of things that can cause an IDS to become suspicious about a connection's intent: unusual destination hosts or ports, scanning behavior by the source host in the past, matches to traffic flow signatures, or a large number of IP fragments are just a small set of examples. Our approach adds another indicator to the toolbox: deviation of the interpretations on the end host and the NIDS can also be used to classify a connection as more suspicious than others, initiating closer scrutiny of such traffic.

IDS Hardening. Lastly, differing interpretations of the same data might simply point out subtle bugs in the implementation of the NIDS, or even in the application itself.

More generally, we see that there are two—somewhat complementary— approaches to leveraging host-supplied context. First, the host can provide *additional* context for the NIDS to include into its analysis. Second, the host can supply *redundant* context which the NIDS uses to verify information is has distilled itself.

5.7.2 Integration into Bro

We incorporate host-supplied context into the Bro system's independent state architecture. To this end, we enable selected applications to send events to a central Bro instance. Similar to Bro's core-generated events, remote events still represent policy neutral descriptions of phenomena occurring within individual process executions. This implies that the policy that determines the relevance of these events is exclusively maintained on the Bro host. The benefits of maintaining the policy here, rather than pushed out to the end hosts, are twofold: first, the policy is accessible centrally and thus easier to adapt; second, this approach imposes less overhead on the monitored host than ordinary HIDSs since the data is not analyzed on the host itself. Generating and sending an event does not cost the host much more effort than writing to a log file.

To interface other applications to Bro, we use Broccoli, the client-side library which implements Bro's independent state mechanisms for use by other applications (see §5.4.4). Using Broccoli, we can instrument a host process with fairly little effort.

We do not make any assumptions about the semantics of remote events. Usually, their meaning is application-specific. However, different applications may generate the same kind of events. For example, a Web server and an HTTP proxy may both communicate URLs. If suitable, remote events may also directly map to some of Bro's internal events. In this case, the default processing of such events can be leveraged.

Bro's connection-oriented view of traffic analysis raises significant issues for the integration of remote events with existing local state. Essentially, we need to unite the stream of events generated by observing a connection on the wire with the stream of events generated by the remote application that processes the connection's data. One avenue for doing so is to have the remote application send along the parameters identifying the connection, for example the IP/port quadruple. In order for this to work, the analyzer must be structured in a way to allow a fusion of event streams. This means that we must make available all state required to process the events to all relevant event handlers. Furthermore, this state must be structured to support the processing of events of different origins and levels of abstraction levels. One instance of this problem space is the need for synchronization when we cannot guarantee that the Bro host can monitor all relevant traffic: we must ensure that new state can be instantiated by both local and remote events, and that this state is not expired prematurely.

5.7.3 Examining HTTP Sessions

As a case study, we decided to take a closer look at HTTP, the most widely used application-layer protocol in the Internet. It is not uncommon that Web traffic amounts for more than half of all TCP connections in a large network. All major NIDSs provide components to detect HTTP-based attacks, which at a minimum extract the requested URLs from the network stream and match these against a set of signatures to detect malicious requests.

The main observation here is that there are at least two HTTP decoders which dissect the same HTTP connection, namely the Web server and the NIDS. While this is a

duplication of work, the separation of the tasks is indeed reasonable: per our discussion above, we prefer the Web server not to perform the intrusion detection itself (and, naturally, it does not make sense for the NIDS to serve HTTP requests). However, this redundancy allows us to benefit from both additional and redundant context, as discussed in §5.7.1. We now discuss both approaches in turn. While we focus on URLs extracted from the requests, we note that similar reasoning holds for deeper inspection.

Leveraging Additional Web Server Context

With respect to the semantics of a given HTTP request, it is obviously the Web server that is authoritative: its environment-specific configuration defines the interpretation of the request and the meaning of any reply. Thus, providing the NIDS with information from the Web server promises to offer a significant increase in contextual information (see §2.3.7).

Web servers can provide several kinds of context that are hard or impossible for the NIDS to derive by itself:

Decryption. SSL-enabled sessions have become quite common for transferring sensitive data. While quite desirable, this poses severe restrictions on passive application-layer network monitoring. However, since the Web server decrypts such requests, it can provide them as clear-text to the NIDS via an independent (and again encrypted) channel.

Full Request Processing. The Web server always fully decodes the request stream it receives. In contrast, many NIDSs perform this task rather half-heartedly; e.g., Snort [Roe99] may miss requests in pipelined/persistent connections if they cross packet boundaries. Older versions used to extract only the very first URL from each packet.

Full Reply Processing. Some information can be easily provided by the Web server while a NIDS needs to put considerable effort into deriving it. For example, Bro is able to extract the server's reply code from HTTP sessions. But this comes at a prohibitive processing cost. On the other hand, for the Web server there is no additional cost involved in providing the result, other than that of sending the data to the NIDS.

Disambiguation. The document eventually served can substantially differ from the one requested. The server resolves the path inside an URL in a virtual namespace; without further context it may not be predictable which *file* is returned in response. Redirection and rewriting mechanisms internal to the server can change the URL path arbitrarily. For a NIDS to follow the exact same steps as the Web server, it would need to know all related configuration statements as well as the full file system layout of the Web server — infeasible in practical terms. Furthermore, most NIDSs are simply not flexible enough to accommodate such a "shadow configuration".

Avoiding Evasion using Redundant Context

Evasion attacks (see §2.4.3) can be used to mislead the NIDS's HTTP protocol decoding. If the NIDS extracts a different HTTP request than the Web server—or if it does not see one at all—it may produce both false negatives and false positives. However, if we can compare the outcome of the two HTTP decoders, we have an opportunity to detect these mismatches.

For a Web session, network- and transport-layer evasion attacks can be used to hide, alter, or inject URLs. Moreover, there are ways to evade the application-layer HTTP decoders of NIDSs. The most prevalent form is *URL encoding* [HM04]. Per RFC 2396 [BFIM98], URLs may only contain a subset of the US-ASCII characters. However, to represent other characters, arbitrary values can be encoded using special control sequences. For example, Web servers are required to support the "percent-encoding" which can encode arbitrary hexadecimal values. Some Web servers—most notably Microsoft's IIS—also provide more sophisticated encodings, such as Unicode [Roe04].

For a NIDS, it is hard to precisely mimic these encodings and character sets. In the past, many systems required fixes upon the discovery of new encoding tricks (e.g., [Int01]). In general, a Web server's eventual interpretation of an URL depends on its local environment and configuration, making it nearly impossible for a NIDS to derive it. This issue is part of the more general problem of NIDSs often lacking context required to reliably detect attacks (see §6).

Often, such application-layer encoding attacks do in fact not target the NIDS but the Web server itself. Due to implementation bugs, such an encoding may circumvent server-internal sanity checks. For example, `CVE-2001-0333` [CVE01a] discusses a flaw in the IIS server which leads to a filename being decoded twice. We can detect such bugs if we compare the decoding the Web server performs with the independent result of the NIDS. Similarly, the NIDS might have flaws that show up when verified with the outcome of the Web server.

Finally, while comparing the output of the two decoders can detect both evasion attacks and implementation flaws, we must also prepare ourselves for the possibility of numerous *benign* differences, which we explore further below.

5.7.4 Deployment and Results

For our case study, we have evaluated our approach in three installations: an experimental testbed and two production environments. All use the Apache Web server and the Bro NIDS.

Setup

We instrumented the Apache Web server with a Broccoli client that communicates with an instance of the Bro NIDS running concurrently on either the same machine or a remote host. The communication between Apache and Bro is one-way. For each request, Apache sends the involved hosts and TCP ports, the original request string, the URL as canonicalized by Apache, the name of the file being served, and the HTTP reply code.

This information is available through Apache's default logging module (except we need a slight extension to access the ports).

There are two different ways of connecting the Web server with Broccoli. The first, which is particularly unobtrusive, is using a separate process for the Broccoli client, which either reads the Apache log file (i.e., no modification to Apache at all) or communicates with Apache via a pipe (i.e., only a small change to Apache's configuration). The second is to integrate Broccoli directly into Apache. We implemented both of these. We used the first for our operational/testbed deployments, and the second for our performance testing (detailed below).

When Bro receives the information from an Apache request, it runs two kinds of analysis, corresponding to the two main uses identified in §5.7.3. First, it passes the canonicalized URL through its standard detection process. This includes both script-layer analysis and event-layer signature matching. Second, it matches the URL against the one extracted by Bro itself from the connection's packet stream. If it encounters a difference, it generates an alert.

In our testbed, we installed Apache 2.0.52 and a recent development version of Bro on the same host. We let Bro run its default HTTP analysis on the packet stream as seen on the loopback device. The Apache-supplied information was sent over a TCP connection from the Broccoli client to the Bro system.

We also instrumented two production Web servers at TUM: the main Web server of the the Computer Science Department, and the server of the Network Architectures Group, located inside WGN. The main server handles between 20,000 and 30,000 requests per day. To monitor it, we used the approach of a separate Broccoli client reading from the log file. The Network Architecture Group's server processes about 5,000-6,000 requests a day. For it, we ran Bro on the same host and used a pipe between Apache and the Broccoli client, like we did with the testbed.

Experiences

We operated these setups for two weeks, with very encouraging results. We first discuss how the additional context indeed provides significant benefits for the detection process, and then our preliminary experiences with evaluating redundant context to detect evasion attacks and decoding flaws. We also note that maintaining the analysis policy on the Bro side while keeping the Broccoli client policy-neutral proved valuable: we could change the configuration of the NIDS at will without needing to touch the Web servers.

Additional Context. Incorporating context supplied by Apache proved to be a major gain. First, we could confirm that the NIDS reliably saw all requests served by the Web server—a major benefit, since in high-volume environments a NIDS running on commodity hardware regularly drops packets and therefore may miss accesses (see §4).

Next, we confirmed that Bro could perform signature matching on the URLs and filenames even if we omitted HTTP decoding from Bro's configuration. For high-volume Web servers, this holds the potential to realize a major performance gain,

since HTTP analysis can easily increase total CPU usage by a factor of 4–6 (see §4.4.4).

Bro's signature engine assumes that internal connection state already exists when matching signatures for a given connection. But if Bro is not decoding the HTTP traffic directly, but only receiving it as a feed from Apache, it will not have instantiated this state. Fortunately, we can arrange for Bro to instantiate such state by having it capture TCP control packets (SYNs, FINs and RESETs). In our experience, it is quite feasible to analyze *all* such control packets even in highly loaded Gbps environments. Note, though, that this approach limits internal signature matching to HTTP sessions which Bro sees itself. Matching on requests from unseen connections (for example, those internal to the site) will require additional internal modifications which are part of a larger plan to change some aspects of Bro's core. Also, we note that this restriction only applies to the internal signature engine. Script-level analysis, such as regular expression matching, is generally possible even without internal connection state.

Bro uses bidirectional signatures to avoid false positives (see §6.2.2). For example, many of the HTTP signatures only alert if the server does not respond with an error message. Since Apache supplies us with its reply code as well, we retain this important feature.

Finally, for the first time we are able to detect attacks in SSL-encrypted sessions. We verified that Bro indeed received the decrypted information and spotted sensitive accesses within them.

Redundant Context. We configured the Bro system to automatically compare the URLs received from the Apache server with those distilled by its own HTTP decoder. There are cases in which differences in the URLs are legitimate. Most importantly, the Web server may internally expand the requested URL, for example when expanding a request like `/foo/bar/` into `/foo/bar/index.html`. However, from our preliminary experiences with the two production servers, it appears that in practice such differences may be rare enough to be explicitly coded into the NIDS's configuration. Consequently, for Bro we implemented an expansion table of regular expressions that reproduces such URL translations.

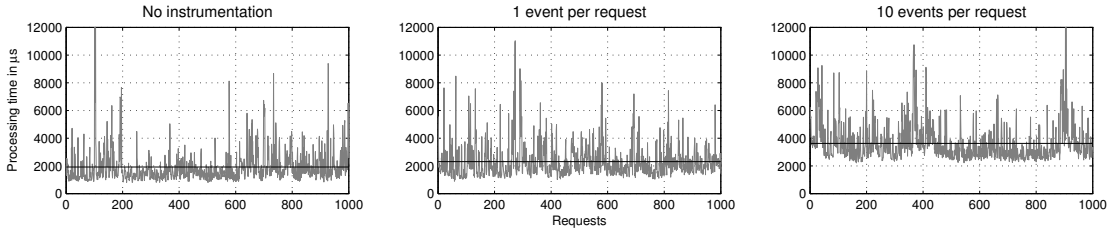
Before we compare two URLs, we also strip CGI parameters. When logging a URL, Apache does not remove the URL-encoded parameters. Bro, on the other hand, decodes the parameters fully. Therefore, such stripping is required to avoid mismatches in accesses to CGI scripts.

This policy was running well on our production servers. The main source of differences we encountered were with requests of the form

```
+GET http://www.foo.bar/index.html HTTP/1.x
```

Such requests indicate that somebody is trying to use the Web server as a proxy. Apache strips `http://www.foo.bar` before processing the request; Bro does not.

Figure 5.8 Overhead of Bro event transmission.



Examining these requests more closely revealed that these were mostly scans for open proxies. Others indicated client misconfigurations.

We found additional differences between Apache and Bro. None of these turned out to be security-relevant (e.g., we saw client requests which included labels of the form "foo.html#label"; these labels are removed by Apache). However, the question remains whether in a larger-scale environment such differences would occur often enough, and in sufficiently varied forms, to significantly complicate the use of redundant context for detecting evasion attempts and decoder flaws.

To stress both Apache and Bro more intensively, we installed three evasion tools in our test-lab. *Libwhisker* [LW] is a Perl library which includes various URL encoding tricks supposed to evade NIDSs or the security mechanisms of a Web server [Pup99]. It includes a command-line script for issuing individual requests to a server. We patched this script to selectively enable one or more of the evasion methods. We also installed the penetration testing tool *Nikto* [Nik], which ships with a large library of HTTP requests to exploit known server vulnerabilities. Internally, Nikto leverages libwhisker. Therefore, it is able to encode its requests using libwhisker's evasion techniques. Finally, we used a small stand-alone encoder [Roe], which converts arbitrary strings into different Unicode representations.

The results of our evasion experiments are encouraging. Both systems, Apache and Bro, decode the crafted requests without any hitch, yet with the following differences:

- Libwhisker can insert relative directory references into the URLs, turning `/foo/bar/` into, e.g., `/foo/./bar/` or `/garbage/../foo/bar/`. Apache canonicalizes the path. Bro leaves it untouched, which for a NIDS not knowing the Web server's filesystem layout makes sense: subsequent analysis may want to alert on these references.
- To avoid ambiguities, double-encoded requests are never to be decoded more than once. (In a double encoding, a character such as 'z' — ASCII `0x7a` — is encoded as `%%37%41`. The first decoding step yields `%7a`, then the second gives '`z`'). If Apache encounters such a request, it logs the result of the first decoding step but sends an error to the client. Bro also decodes it only

once, but removes the additional percentage sign before further processing. In addition, it reports the ambiguity. While their behaviors differ, both systems recognize the situation and report an error.

- Requests containing Unicode characters (literally, or encoded with the IIS-proprietary `%u` encoding) are either left untouched or treated as an error by Apache.[11] Bro always leaves such characters untouched. Thus, either the two systems agree, or Apache does not serve any document.

To summarize, we see that Apache and Bro appear to work well together in terms of HTTP URL-canonicalization. If in the future we encounter more mismatches, we can now detect them as soon as they occur. We note that our results may not readily apply to other Web servers. For example, Microsoft's IIS supports a handful of other encodings [Roe04] not supported by Bro. In particular, Bro does not include a Unicode decoder yet. In addition, past experience with IIS vulnerabilities suggests that its more complex decoder may also be more vulnerable than Apache's.

Performance Evaluation

A key question is whether the performance overhead of the instrumentation is tolerable. We tested the performance impact incurred on Apache using *httperf* [MJ98] as a load generator. We ran each of httperf, Apache, and Bro on separate machines (2.53 Ghz Pentium 4s with 500 MB RAM) connected on a 100 Mb/s network. For these measurements, we implemented the Broccoli client in the form of an Apache 1.3 logging module, `mod_bro`, requiring only an additional 120 lines of C code.

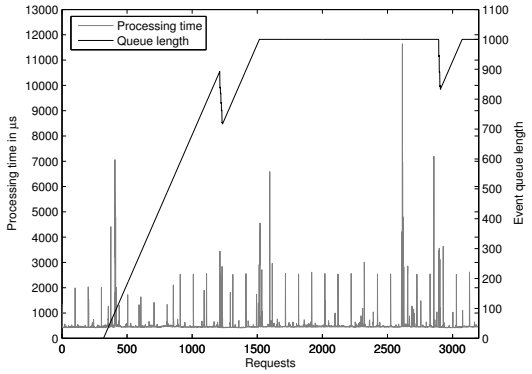
We first measured the per-request overhead of sending Bro events from a lightly loaded Apache. We requested a single, static Web page 1,000 times at a rate of 20 connections per second, measuring the request processing times using the `mod_benchmark` module [Mod], and averaged the results of the nth request across 10 separate runs. The results are shown in Figure 5.9: on average, Apache required around 2ms for each request. Sending the single Bro event necessary for our contextual analysis had a minimal performance impact, in the order of 300 μs per request.

The second experiment tested the overhead with a Bro under heavy load. To emulate this situation reliably, we artificially introduced a processing delay of 0.2 s per received event on the Bro side.[12] Broccoli clients have a bounded per-connection event queue that we configured to a maximum size of 1,000 events. Additional events enqueued at this point lead to the oldest events being dropped. To simplify the queuing behavior, we ran Apache with a single process serving requests only. The results are shown in Figure 5.9: the workload of the receiving Bro host does not noticeably affect the application's performance.

[11] This is true for Unix systems. On Windows, Apache may handle Unicode differently but we have not examined this further.

[12] 0.2 s turned out to be a suitable value, causing a reproduceable queue build-up.

Figure 5.9 Overhead of Broccoli event transmission.



In our production installations we always connected a single Web server to Bro. To explore how our setup might scale with more instrumented servers, we measured the amount of data exchanged between one instance of Apache and the receiving Bro. This volume depends on the number of HTTP requests as well as the length of the requested URLs, but is independent of the HTTP connection's actual payload size. A single run of Nikto (see §5.7.4) issues 2,443 requests to the Web server. On average, for every request 455 bytes of payload are transmitted between Apache and Bro.[13] Thus, the network load is modest: under 1 Mbps for 2,000 requests/sec, a level that can accommodate a good number of busy Web servers. For the Bro side, the amount of work to process the received bytes is, in general, much less than to parse the full HTTP stream (for Bro's HTTP processing, the experiments performed in §4.4.4 showed a performance decrease of a factor of 4-6). Therefore, one option here is to significantly lighten the load on Bro by leveraging the Web server's processing and context, which should enable Bro's monitoring to scale to significantly higher HTTP loads than before.

To summarize, from our preliminary assessment the overhead imposed by instrumenting applications to participate in the event communication of a network of Bro nodes appears quite acceptable.

5.8 Summary

In this chapter, we demonstrated the power of exploiting *independent state* in network intrusion detection. While traditionally much of a NIDS's state resides solely in volatile memory, we argue for making all of a NIDS's state exist (potentially) "outside" of any particular process. To this end, we developed the notions of *spatially independent state*

[13] Roughly two thirds of these bytes come from protocol overhead. While high, we remind that Bro's communication protocol can exchange serializations of Bro's complex data structures while ensuring type-safety, reconstructing reference structures, and performing architecture-independent data marshaling (see §5.4.1). We thus trade off efficiency for flexibility here.

(state that can be propagated from one instance of the NIDS to another concurrently running process) and *temporally independent state* (state that continues to exists after the termination of all instances, available to future processes). The architecture we implemented facilitates independent state for the Bro NIDS. It is *unified* in that it encompasses all of the internal, fine-grained state of the NIDS. Thereby, we can continue to process independent state using the full set of mechanisms provided by the system.

The main internal mechanism of our architecture is a *serialization framework*. While its implementation was straight-forward in general, the system's internal complexity gave us a number of subtle issues to solve. Having the serialization in place, we added a user-level interface driven by our operational applications. It enables the user to selectively declare state to be independent. To achieve temporal independence, we serialize state into files, either when an instance exits, or incrementally as it executes. A subsequent process can then read it back. To achieve spatial independence, we added secure network communication to the NIDS, allowing instances to share state across different locations.

The architecture provides us with a wealth of possible applications. We enhanced Bro's traditional model of regular checkpointing by allowing a *controlled* loss of state, added crash-recovery, examined different approaches for distributing the monitoring and analysis, enabled run-time policy management, and greatly extended the system's profiling and debugging facilities. These applications were driven by our operational experiences, and we experimented with all of them in our environments. A performance evaluation of the communication component showed that our implementation is suitable for deployment even in large-scale installations.

We demonstrated the broader notion of independent state by sharing state with non-NIDS components: we included host-based application context into Bro's detection process. A NIDS can use such contextual information to supplement/verify its analysis. To assess the feasibility of this approach, we instrumented the Apache Web server with an interface to Bro, and installed the Apache/Bro combo in two production environments as well as in a testbed. The results from these deployments are encouraging and emphasize the power of the independent state concept.

6 Enhancing Signatures with Context

> 'Serpent!' screamed the Pigeon. 'I'm not a serpent!' said Alice indignantly. [...] 'No, no! You're a serpent; and there's no use denying it. I suppose you'll be telling me next that you never tasted an egg!' 'I have tasted eggs, certainly,' said Alice, who was a very truthful child; 'but little girls eat eggs quite as much as serpents do, you know.' 'I don't believe it,' said the Pigeon; 'but if they do, why then they're a kind of serpent, that's all I can say.'
>
> — *Lewis Carroll, "Alice's Adventures in Wonderland"*

Most NIDSs use signatures to detect malicious activity. While being highly efficient, they tend to suffer from a high false-positive rate. In high-performance environments, this is major problem. Due to the high traffic volume/variance, there are often much more alerts triggered than the operators can realistically track down. In this chapter, we develop the concept of *contextual signatures* as an improvement of string-based signature-matching to reduce the number of false positives. Rather than matching fixed strings in isolation, we augment the matching process with additional context. When designing an efficient signature engine for the Bro NIDS, we provide low-level context by using regular expressions for matching, and high-level context by taking advantage of the semantic information made available by Bro's protocol analysis and scripting language. Therewith, we greatly enhance the signature's expressiveness and hence the ability to reduce false positives. We present several examples such as matching requests with replies, using knowledge of the environment, defining dependencies between signatures to model step-wise attacks, and recognizing exploit scans. To leverage existing efforts, we convert the signature set of the Snort NIDS into Bro's language. While this does not provide us with improved signatures by itself, we reap an established base to build upon. We evaluate our work by comparing it to Snort.

6.1 Overview

In this chapter, we concentrate on one popular form of misuse detection, *signature matching* (see §2.3.4). Signature-matching has several appealing properties. First, the underlying conceptual notion is simple: it is easy to explain what the matcher is looking for and why, and what sort of total coverage it provides. Second, because of this simplicity, signatures can be easy to share, and to accumulate into large "attack libraries."

Third, for some signatures, the matching can be quite *tight*: a match indicates with high confidence that an attack occurred.

On the other hand, signature-matching also has significant limitations. In general, especially when using tight signatures, the matcher has no capability to detect attacks other than those for which it has explicit signatures; the matcher will in general completely miss novel attacks, which, unfortunately, continue to be developed at a brisk pace. In addition, as we discuss in §4.2.1, often signatures are not in fact tight. Loose signatures immediately raise the major problem of false positives: alerts that do not reflect an actual attack. A second form of false positive, which signature matchers likewise often fail to address, is that of *failed attacks*. Since at many sites attacks occur at nearly-continuous rates, failed attacks are often of little interest. At a minimum, it is important to distinguish between them and successful attacks.

A key point here is that the problem of false positives can potentially be greatly reduced if the matcher has additional *context* at its disposal: either additional particulars regarding the exact activity and its semantics, in order to weed out false positives due to overly general "loose" signatures; or the additional information of how the attacked system responded to the attack, which often indicates whether the attack succeeded.

In this chapter, we develop the concept of *contextual signatures*, in which the traditional form of string-based signature matching is augmented by incorporating additional context on different levels when evaluating the signatures. First of all, we design and implement an efficient pattern matcher similar in spirit to traditional *signature engines* used in other NIDS. But already on this low-level we enable the use of additional context by *(i)* providing full regular expressions instead of fixed strings, and *(ii)* giving the signature engine a notion of full connection state, which allows it to correlate multiple interdependent matches in both directions of a user session. Then, if the signature engine reports the match of a signature, we use this event as the *start* of a decision process, instead of an alert by itself as is done by most signature-matching NIDSs. Again, we use additional context to judge whether something alert-worthy has indeed occurred. This time the context is located on a higher-level, containing our knowledge about the network that we have either explicitly defined or already learned during operation.

We show several examples to demonstrate how the concept of contextual signatures can help to eliminate most of the limitations of traditional signatures discussed above. We see that regular expressions, interdependent signatures, and knowledge about the particular environment have significant potential to reduce the false positive rate and to identify failed attack attempts. For example, we can consider the server's response to an attack and the set of software it is actually running—its *vulnerability profile*—to decide whether an attack has succeeded. In addition, treating signature matches as events rather than alerts enables us to analyze them on a meta-level as well, which we demonstrate by identifying *exploit scans* (scanning multiple hosts for a known vulnerability).

We implemented the concept of contextual signatures in the framework already provided by the Bro NIDS. One motivation for this work is to combine Bro's flexibility with the capabilities of other NIDSs by implementing a signature engine. But in contrast to traditional systems, which use their signature matcher more or less on its own, we tightly integrate it into Bro's architecture in order to provide contextual signatures. Second,

on a higher level, we use Bro's rich contextual state to implement our improvements to plain matching. Making use of Bro's architecture, our engine sends events to the policy layer. There, the policy script can use all of Bro's already existing mechanisms to decide how to react.

Due to Snort's large user base, it provides a comprehensive set of signatures. Therefore, although for flexibility we designed a custom signature language for Bro, we make use of the Snort libraries via a conversion program. Even if we do not accompany them with additional context, they immediately give us a baseline of already widely-deployed signatures. Yet, we need to keep their general lack of quality in mind (see §4.2.2) which we confirmed when using converted Snort signatures.

In general, Snort serves us as a reference. Throughout the chapter, we compare with Snort both in terms of quality and performance. While doing so, we encountered several of the general problems for evaluating and comparing NIDSs. We believe that these arise independently of our work with Bro and Snort, and therefore describe them in some detail. Keeping these limitations in mind, we then evaluate the performance of our signature engine and find that it performs well.

We note that this work was conducted when Snort 1.9.0 was current. In its current version, Snort's matching engine has seen major improvements. Most importantly, Snort is now able to match regular expressions in addition to traditional-style fixed strings. However, in contrast to our engine, it uses the brute-force approach of matching each regular expression individually. To avoid performance problems, Snort's manual explicitly suggests to have at least one fixed-string condition in each signature.

In §6.2 we present the main design ideas behind implementing contextual signatures: regular expressions, integration into Bro's architecture, and examples of the power of the Bro signature language. In §6.3 we compare Bro's signature matching with Snort's. We report our operational experiences with using contextual signatures in §6.3.4. Finally, §6.4 summarizes our conclusions.

6.2 Contextual Signatures

The heart of Bro's contextual signatures is a signature engine designed with three goals in mind: *(i)* expressive power, *(ii)* the ability to improve alert quality by utilizing Bro's contextual state, and *(iii)* enabling the reuse of existing signature sets. We discuss each in turn. Afterwards, we show examples which demonstrate the capabilities of the described concepts.

6.2.1 Regular Expressions

A traditional signature usually contains a sequence of bytes that are representative of a specific attack. If this sequence is found in the payload of a packet, it is an indicator of a possible attack. Therefore, the matcher is a central part of any signature-based NIDS. While many NIDSs only allow fixed strings as search patterns, we argue for the utility of using *regular expressions*. Regular expressions provide several significant advantages: first, they are far more flexible than fixed strings. Their expressiveness

has made them a well-known tool in many applications, and their power arises in part from providing additional *syntactic context* with which to sharpen textual searches. In particular, character classes, union, optional elements, and closures prove very useful for specifying attack signatures, as we see in §6.2.4.

Surprisingly, given their power, regular expressions can be matched very efficiently. This is done by compiling the expressions into DFAs whose terminating states indicate whether a match is found. A sequence of n bytes can therefore be matched with $O(n)$ operations, and each operation is simply an array lookup—highly efficient.

The total number of patterns contained in the signature set of a NIDSs can be quite large. Snort's set, for example, contained 1,715 distinct signatures in version 1.9.0, of which 1,273 were enabled by default. Matching such a number individually is very expensive. However, for fixed strings, there are algorithms for matching sets of strings simultaneously. Consequently, while Snort's default engine used to work iteratively for a long time, work to replace it with a "set-wise" matcher was on its way early [CSM01, FV01].[1] On the other hand, regular expressions give us set-wise matching for free: by using the union operator on the individual patterns, we get a new regular expression which effectively combines all of them. The result is a single DFA that again needs $O(n)$ operations to match against an n byte sequence. Only slight modifications were necessary to extend the interface of Bro's already-existing regular expression matcher to explicitly allow grouping of expressions.

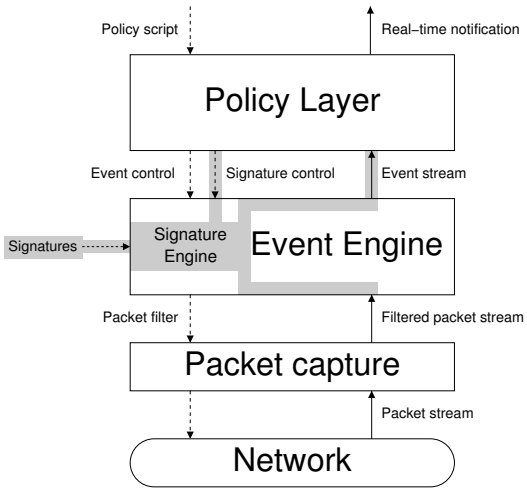
Given the expressiveness and efficiency of regular expressions, there is still a reason why a NIDS might avoid using them: the underlying DFA can grow very large. Fully compiling a regular expression into a DFA leads potentially to an exponential number of DFA states, depending on the particulars of the patterns [HU79]. Considering the very complex regular expression built by combining all individual patterns, this straightforward approach can easily become intractable. Our experience with building DFAs for regular expressions matching many hundreds of signatures shows that this is indeed the case. However, it turns out that in practice it is possible to avoid the state/time explosion, as follows.

Instead of pre-computing the DFA, we build the DFA "on-the-fly" during the actual matching [HKR92]. Each time the DFA needs to transit into a state that is not already constructed, we compute the new state and record it for future reuse. This way, we only store DFA states that are actually needed. An important observation is that for n new input characters, we will build at most n new states. Furthermore, we find in practice (§6.3.3) that for normal traffic the growth is *much* less than linear.

However, there is still a concern that given inauspicious traffic—which may actually be artificially crafted by an attacker—the state construction may eventually consume more memory than we have available. Therefore, we also implemented a memory-bounded DFA *state cache*. Configured with a maximum number of DFA states, it expires old

[1] The code of [FV01] was already contained in the Snort 1.9.0 distribution, but not compiled-in by default. This was perhaps due to some subtle bugs, some of which we encountered during our testing as well.

Figure 6.1 Integrating the signature engine into Bro's architecture.



states on a least-recently-used basis. In the sequel, when we mention "Bro with a limited state cache," we are referring to such a bounded set of states (which is a configuration option for our version of Bro), using the default bound of 10,000 states.

Another important point is that it's not necessary to combine all patterns contained in the signature set into a *single* regular expression. Most signatures contain additional constraints like IP address ranges or port numbers that restrict their applicability to a subset of the whole traffic. Based on these constraints, we can build groups of signatures that match the same kind of traffic. By collecting only those patterns into a common regular expression which are in the same group, we are able to reduce the size of the resulting DFA drastically. As we show in §6.3, this gives us a very powerful pattern matcher still efficient enough to cope with high-volume traffic.

6.2.2 Improving Alert Quality by Using Context

Though pattern matching is a central part of any signature-based NIDSs, there is potentially great utility in incorporating more context in the system's analysis prior to generating an alert, to ensure that there is indeed something alert-worthy occurring. We can considerably increase the quality of alerts, while simultaneously reducing their quantity, by utilizing knowledge about the current state of the network. Bro is an excellent tool for this as it already keeps a lot of easily accessible state.

The new signature engine is designed to fit nicely into Bro's layered architecture as an adjunct to the protocol analysis event engine (see Figure 6.1). We implemented a custom language for defining signatures. It is mostly a superset of other, similar languages, and we describe it in more detail in §6.2.3. A new component placed within Bro's middle layer matches these signatures against the packet stream. Whenever it finds a match, it inserts an event into the event stream. The policy layer can then decide how to react.

Adding signatures to the Bro system introduces a second configuration language in addition to the policy-layer scripting language. However, while signatures are supplied by the user, we do not picture them to live on the policy-layer as Bro's traditional scripts do. Signatures control the matching process inside the event engine, yet they cannot produce any alerts by themselves; they can only raise events which are then further processed. The policy layer remains in charge of escalating such matches if necessary.

In addition to raising events for matches, we can also pass information from the policy layer back into the signature engine to control its operation. A signature can specify a script function to call whenever a particular signature matches. This function can then consult additional context and indicate whether the corresponding match-event should indeed be generated. We show an example of this later in §6.2.4.

In general, Bro's analyzers follow the communication between two endpoints and extract protocol-specific information. For example, the HTTP analyzer is able to extract URLs requested by Web clients (which includes performing general preprocessing such as expanding hex escapes) and the status code and items sent back by servers in reply, whereas the FTP analyzer follows the application dialog, matching FTP commands and arguments (such as the names of accessed files) with their corresponding replies. Clearly, this protocol-specific analysis provides significantly more context than does a simple view of the total payload as an undifferentiated byte stream.

The signature engine can take advantage of this additional information by incorporating semantic-level signature matching. For example, the signatures can include the notion of matching against HTTP URLs; the URLs to be matched are provided by Bro's HTTP analyzer. Having developed this mechanism for interfacing the signature engine with the HTTP analyzer, it is straight forward to extend it to other analyzers and semantic elements (indeed, we timed how long it took to add and debug interfaces for FTP and Finger, and the two totaled only 20 minutes).

Central to Bro's architecture is its connection management. Each network packet is associated with exactly one connection. This notion of connections allows several powerful extensions to traditional signatures. First of all, Bro reassembles the payload stream of TCP connections. Therefore, we can perform all pattern matching on the actual stream (in contrast to individual packets). While Snort has a preprocessor for TCP session reassembling, it does so by combining several packets into a larger "virtual" packet. This packet is then passed on to the pattern matcher. Because the resulting analysis remains packet-based, it still suffers from discretization problems introduced by focusing on packets, such as missing byte sequences that cross packet boundaries.

In Bro, a signature match does not necessarily correspond to an alert; as with other events, that decision is left to the policy script. Hence, it makes sense to remember which signatures have matched for a particular connection so far. Given this information, it is then possible to specify dependencies between signatures like "signature A only matches if signature B has already matched," or "if a host matches more than N signatures of type C, then generate an alert." This way, we can for example describe multiple steps of an attack. In addition, Bro notes in which direction of a connection a particular signature has matched, which gives us the notion of *request/reply signatures*: we can associate a client request with the corresponding server reply. A typical use is to differentiate between successful and unsuccessful attacks. We show an example in §6.2.4.

More generally, the policy script layer can associate arbitrary kinds of data with a connection or with one of its endpoints. This means that any information we can deduce from any of Bro's other components can be used to improve the quality of alerts. We demonstrate the power of this approach in §6.2.4.

Keeping per-connection state for signature matching naturally raises the question of *state management*: at some point in time we have to reclaim state from older connections to prevent the system from exhausting the available memory. But again we can leverage the work already being done by Bro. Independently of the signatures, it already performs a sophisticated connection-tracking using various timeouts to expire connections (see §4.4). By attaching the matching state to the already-existing per-connection state, we assure that the signature engine works economically even with large numbers of connections.

6.2.3 Signature Language

Any signature-based NIDS needs a language for actually defining signatures. For Bro, we had to choose between using an already existing language and implementing a new one. We decided to create a new language for two reasons. First, it gives us more flexibility. We can more easily integrate the new concepts described in §6.2.1 and §6.2.2. Second, to use existing signature sets, it is easier to write a converter in some high-level scripting language than to implement a parser within Bro itself.

Our signatures are defined by means of an identifier and a set of attributes (see Figure 6.2(b) for an example). There are two main types of attributes: *(i) conditions* and *(ii) actions*. The conditions define when the signature matches, while the actions declare what to do in the case of a match. Conditions can be further divided into four types: *header*, *content*, *dependency*, and *context*.

Header conditions limit the applicability of the signature to a subset of traffic that contains matching packet headers. For TCP, this match is performed only for the first packet of a connection. For other protocols, it is done on each individual packet. In general, header conditions are defined by using a *tcpdump*-like [TCP] syntax (for example, `tcp[2:2] == 80` matches TCP traffic with destination port 80). While this is very flexible, for convenience there are also some short-cuts (e.g., `dst-port == 80`).

Content conditions are defined by regular expressions. Again, we differentiate two kinds of conditions here: first, the expression may be declared with the `payload` statement, in which case it is matched against the raw packet payload (reassembled where applicable). Alternatively, it may be prefixed with an analyzer-specific label, in which case the expression is matched against the data as extracted by the corresponding analyzer. For example, the HTTP analyzer decodes requested URLs. So, the condition `http /(etc\/(passwd|shadow)/` matches any request containing either `etc/passwd` or `etc/shadow`.

Signature conditions define dependencies between signatures. We implemented `requires-signature`, which specifies another signature that has to match on the same connection first, and `requires-reverse-signature`, which additionally requires the match to happen for the other direction of the connection. Both conditions can be negated to match only if another signature does *not* match.

Finally, context conditions allow us to pass the match decision on to various components of Bro. They are only evaluated if all other conditions have already matched. For example, we implemented a `tcp-state` condition that poses restrictions on the current state of the TCP connection, and `eval` which calls a boolean script-layer function and evaluates to true if the return value of the function indicates so. Thereby, `eval` provides a callback-mechanism to implement arbitrary complex conditions via Bro's scripting language.

If all conditions are met, the actions associated with a signature are executed. Currently, the only available action is `event` which inserts a `signature_match` event into the event stream, with the value of the event including the signature identifier, corresponding connection, and other context. The policy layer can then analyze the signature match.

Converting Snort Signatures

Snort's signatures are comprehensive, free for non-commercial use and frequently updated. Therefore, we are interested in converting them into our signature language. We wrote a corresponding Python script that takes an arbitrary Snort configuration and outputs signatures in Bro's syntax.[2] Figure 6.2(a) shows an example of such a conversion.

It turns out to be rather difficult to implement a complete parser for Snort's language. As far as we were able to determine, its syntax and semantics are not fully documented, and in fact often only defined by the source code. In addition, due to different internals of Bro and Snort, it is sometimes not possible to keep the exact semantics of the signatures. We discuss such difficulties in §4.2.5.

6.2.4 Applications

In this section, we show several examples to convey the power which our signatures provide when combined with Bro's existing mechanisms. First, we demonstrate how to define more "tight" signatures by using regular expressions. Then, we show how to identify failed attack attempts by considering the set of software a particular server is running (we call this its *vulnerability profile* and incorporate some ideas from [LWS02] here) as well as the response of the server. We next demonstrate modeling an attack in multiple steps to avoid false positives, and finally show how to use alert-counting for identifying *exploit scans*.

Using Regular Expressions

Regular expressions allow far more flexibility than fixed strings. Figure 6.3(a) shows a Snort signature for `CVE-1999-0172` that generated a large number of false positives at USB's border router (see §3.2.5). Figure 6.3(b) shows a corresponding Bro signature that uses a regular expression to identify the exploit more reliably. `CVE-1999-0172`

[2] We note that our converter does not yet support some of the newer Snort options.

Figure 6.2 Example of converting a signature from Snort to Bro.

```
alert tcp any any -> [a.b.0.0/16,c.d.e.0/24] 80
  ( msg:"WEB-ATTACKS conf/httpd.conf attempt";
    nocase; sid:1373; flow:to_server,established;
    content:"conf/httpd.conf"; [...] )
```

(a) Snort.

```
signature sid-1373 {
  ip-proto == tcp
  dst-ip == a.b.0.0/16,c.d.e.0/24
  dst-port == 80
  # The payload below is actually generated in a
  # case-insensitive format, which we omit here
  # for clarity.
  payload /.*conf\/httpd\.conf/
  tcp-state established,originator
  event "WEB-ATTACKS conf/httpd.conf attempt"
}
```

(b) Bro.

describes a vulnerability of the `formmail` CGI script. If an attacker constructs a string of the form "`...; <shell-cmds>`" (a `|` instead of the `;` works as well), and passes it on as argument of the `recipient` CGI parameter, vulnerable formmails will execute the included shell commands. Because CGI parameters can be given in arbitrary order, the Snort signature has to rely on identifying the `formmail` access by its own. By using a regular expression, we can explicitly define that the `recipient` parameter has to contain a particular character.

Vulnerability Profiles

Most exploits are aimed at particular software, and usually only some versions of the software are actually vulnerable. Given the overwhelming number of alerts a signature-matching NIDS can generate, we may well take the view that the only attacks of interest are those that actually have a chance of succeeding. If, for example, an *IIS* exploit is tried on a Web server running *Apache*, one may not even care. [LWS02] proposes to prioritize alerts based on this kind of vulnerability information. We call the set of software versions that a host is running its *vulnerability profile*. We implemented this concept in Bro. By protocol analysis, it collects the profiles of hosts on the network, using version/implementation information that the analyzer observes. Signatures can then be restricted to certain versions of particular software.

Figure 6.3 Two Snort signatures for CVE-1999-0172.

```
alert tcp any any -> a.b.0.0/16 80
  (msg:"WEB-CGI formmail access";
   uricontent:"/formmail";
   flow:to_server,established;
   nocase; sid:884; [...])
```

(a) Snort using a fixed string.

```
signature formmail-cve-1999-0172 {
  ip-proto == tcp
  dst-ip == a.b.0.0/16
  dst-port = 80
  # Again, actually expressed in a
  # case-insensitive manner.
  http /.*formmail.*\?.*recipient=[^&]*[;|]/
  event "formmail shell command"
}
```

(b) Bro using a regular expression.

As a proof of principle, we implemented vulnerability profiles for HTTP servers which usually characterize themselves via the `Server` header, and for SSH clients and servers which identify their specific versions in the clear during the initial protocol handshake. The software identification is easily extendable to other protocols.

The notion of developing a profile can be extended beyond just using protocol analysis. We could *passively fingerprint* hosts to determine their operating system version information by observing specific idiosyncrasies of the header fields in the traffic they generate, similar to the probing techniques described in [Fyo98], or we could separately or in addition employ *active* techniques to explicitly map the properties of the site's hosts and servers [SP03]. Finally, in addition to automated techniques, we could implement a configuration mechanism for manually entering vulnerability profiles.

Request/Reply Signatures

Further pursuing the idea to avoid alerts for failed attack attempts, we can define signatures that take into account both directions of a connection. Figure 6.4 shows an example. In operational use, we see a lot of attempts to exploit `CVE-2001-0333` to execute the Windows command interpreter `cmd.exe`. For a failed attempt, the server typically answers with a $4xx$ HTTP reply code, indicating an error.[3] To ignore these failed attempts, we first define one signature, `http-error`, that recognizes such replies.

[3]There are other reply codes that reflect additional types of errors, too, which we omit for clarity.

Figure 6.4 Example of a request/reply signature.

```
signature cmdexe-success {
  ip-proto == tcp
  dst-port == 80
  http /.*[cC][mM][dD]\.[eE][xX][eE]/
  event "WEB-IIS cmd.exe success"
  requires-signature-opposite ! http-error
  tcp-state established
}

signature http-error {
 ip-proto == tcp
 src-port == 80
 payload /.*HTTP\/1\.. *4[0-9][0-9]/
 event "HTTP error reply"
 tcp-state established
 }
```

Then we define a second signature, `cmdexe-success`, that matches only if `cmd.exe` is contained in the requested URL (case-insensitive) and the server does *not* reply with an error. It is not possible to define this kind of signature in Snort, as it lacks the notion of associating both directions of a connection.

Attacks with Multiple Steps

An example of an attack executed in two steps is the infection by the *Apache/mod_ssl* worm [CA] (also known as *Slapper*), released in September 2002. The worm first probes a target for its potential vulnerability by sending a simple HTTP request and inspecting the response. If the server identifies itself as Apache, the worm then tries to exploit an *OpenSSL* vulnerability on TCP port 443.

It turns out that the request the worm sends is in fact in violation of the HTTP 1.1 standard [FGM+99] because it does not include a `Host` header, and this idiosyncrasy provides a somewhat "tight" signature for detecting a Slapper probe. Better yet, Figure 6.5 shows two signatures that only report an alert if both steps are performed for a destination that runs a vulnerable OpenSSL version. The first signature, `slapper-probe`, checks the payload for the illegal request. If found, the script function `is_vulnerable_to_slapper` (omitted here) is called. Using the vulnerability profile described above, the function evaluates to true if the destination is known to run Apache as well as a vulnerable OpenSSL version.[4] If so, the signature matches (depending on the configuration this may or may not generate an alert by itself). The header conditions of

[4] Note that it could instead implement a more conservative policy, and return true *unless* the destination is known to not run a vulnerable version of OpenSSL/Apache.

Figure 6.5 Contextual signature for the *Apache/mod_ssl*worm.

```
signature slapper-probe {
  ip-proto == tcp
  dst-ip == x.y.0.0/16 # sent to local net
  dst-port == 80
  payload /.*GET \/ HTTP\/1\.1\x0d\x0a\x0d\x0a/
  eval is_vulnerable_to_slapper # call policy function
  event "Vulner. host possibly probed by Slapper"
}

signature slapper-exploit {
  ip-proto == tcp
  dst-ip == x.y.0.0/16
  dst-port == 443 # 443/tcp = SSL/TLS
  eval has_slapper_probed # test: already probed?
  event "Slapper tried to exploit vulnerable host"
}
```

the second signature, `slapper-exploit`, match for any SSL connection into the specified network. For each, the signature calls the script function `has_slapper_probed`. This function generates a signature match if `slapper-probe` has already matched for the same source/destination pair. Thus, Bro alerts if the combination of probing for a vulnerable server, plus a potential follow-on exploit of the vulnerability, has been seen.

Exploit Scanning

Often attackers do not target a particular system on the Internet, but probe a large number of hosts for vulnerabilities (which we refer to as *exploit scanning*). Such a scan can be executed either *horizontally* (several hosts are probed for a particular exploit), *vertically* (one host is probed for several exploits), or both. While, own their own, most of these probes are usually low-priority failed attempts, the scan itself is an important event. By simply counting the number of signature alerts per source address (horizontal) or per source/destination pair (vertical), Bro can readily identify such scans. We implemented this with a policy script which generates alerts like:

```
a.b.c.d triggered 10 signatures on host e.f.g.h
i.j.k.l triggered signature sid-1287 on 100 hosts
m.n.o.p triggered signature worm-probe on 500 hosts
q.r.s.t triggered 5 signatures on host u.v.x.y
```

6.3 Evaluation

Our approach for evaluating the effectiveness of the signature engine is to compare it to Snort in terms of run-time performance and generated alerts, using semantically equivalent signature sets. We note that, at this point, we do not systematically evaluate the concept of conceptual signatures by itself. Instead we validate that our implementation is capable of acting as an effective substitute for the most-widely deployed NIDS even when we do not use any of the advanced features it provides. In §6.3.4 we then report our experiences with using contextual signatures operationally.

During our comparison of Bro and Snort, we experienced several of the peculiarities discussed in §4.2.5. Our results stress that the performance of a NIDS can be very sensitive to semantics, configuration, input, and even underlying hardware. Therefore, after discussing our test data, we delve into these in some detail. Keeping these limitations in mind, we then assess the overall performance of the Bro signature engine.

6.3.1 Data

For our testing, we used two traces:

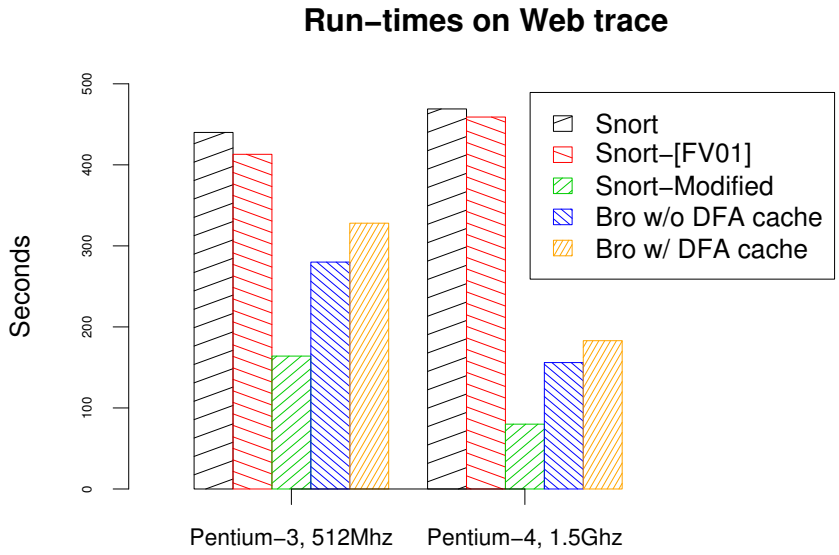
`USB-Full`. A 30-minute trace collected at USB, consisting of all traffic (including packet contents) except for three high-volume peer-to-peer applications (to reduce the volume). The trace totals 9.8 GB, 15.3 M packets, and 220 K connections. 35% of the trace packets belong to HTTP on port 80, 19% to eDonkey on port 4662, and 4% to SSH on port 22, with other individual ports being less common than these three (and the high-volume peer-to-peer that was removed).

`LBL-Web`. A two-hour trace of HTTP client-side traffic, including packet contents, gathered at LBNL. The trace totals 667 MB, 5.5 M packets, and 596 K connections.

We used the version 1.9 branch of Snort, and version 0.8a1 of Bro extended with the our signature engine. Unless stated otherwise, we performed all measurements on 550 MHz Pentium 3 systems containing ample memory (512 MB or more). For both Snort's and Bro's signature engines, we used Snort's default signature set. We disabled Snort's "experimental" set of signatures as some of the latest signatures were using rather new options which, at that point, were not implemented in our conversion program. In addition, we disabled Snort signature #526, *BAD TRAFFIC data in TCP SYN packet*. Due to Bro matching stream-wise instead of packet-wise, it generates thousands of false positives. In total, 1,118 signatures are enabled. They contain 1,107 distinct patterns and cover 89 different service ports. 60% of the signatures cover HTTP traffic. For `LBL-Web`, only these were activated.

For Snort, we enabled the preprocessors for IP defragmentation, TCP stream reassembling on its default ports, and HTTP decoding. For Bro, we turned on TCP reassembling for the same ports (even if otherwise Bro would not reassemble them because none of the usual event handlers indicated interest in traffic for those ports), enabled its memory-saving configuration ("`@load reduce-memory`"), and used an inactivity timeout of 30

Figure 6.6 Processing times of signature matching on different hardware.



seconds (in correspondence with Snort's default session timeout). We configured both systems to consider all packets contained in the traces.

6.3.2 Difficulties

In §4.2.5 we discuss difficulties we encountered when comparing the performance of different NIDSs. While evaluating contextual signatures, we experienced a variety of such problems. Particularly illuminating were the difference in run-time that we saw depending on input and underlying hardware.

On a Pentium 3 system, we run Snort on the trace `LBL-Web`; it needed 440 CPU seconds (see Figure 6.6). This decreased only by 6% when using the set-wise pattern matcher of [FV01]. Yet, we devised a small modification to Snort that, compared to the original version, speeded it up by factor of 2.6 for this particular trace. (The modification is an enhancement to the set-wise matcher: the original implementation first performs a set-wise search for all of the possible strings, caching the results, and then iterates through the lists of signatures, looking up for each in turn whether its particular strings were matched. Our modification uses the result of the set-wise match to identify potential matching signatures directly if the corresponding list is large, avoiding the iteration.)

Using the trace `USB-Full`, however, the improvement realized by our modified set-wise matcher for Snort was only a factor of 1.2. Even more surprisingly, on another trace from NERSC, the original version of Snort was *twice as fast* as the set-wise implementation of [FV01] (148 CPU secs vs. 311 CPU secs), while our patched version lied in between (291 CPU secs). While the reasons remain to be discovered in Snort's internals, this demonstrates the difficulty of finding representative traffic as proposed, e.g., in [HW02].

Furthermore, relative performance does not only depend on the input but even on the underlying hardware. As described above, the original Snort needed 440 CPU seconds for `LBL-Web` on a Pentium 3 based system. Using exactly the same configuration and input on a Pentium 4 based system (1.5 GHz), it actually took 29 CPU seconds *more*. But now the difference between the stock Snort and our modified version is a factor of 5.8! On the same system, Bro's run-time *decreased* from 280 to 156 CPU seconds.

Without detailed hardware-level analysis, we can only guess why Snort suffered from the upgrade. To do so, we ran *valgrind*'s [Val] cache simulation on Snort. For the second-level data cache, it showed a miss-rate of roughly 10%. The corresponding value for Bro was below 1%. While we do not know if valgrind's values are airtight, they could be the start of an explanation. We heard other anecdotal comments that the Pentium 4 performs quite poorly for applications with lots of cache-misses. On the other hand, by building Bro's regular expression matcher incrementally, as a side effect the DFA tables will wind up having memory locality that somewhat reflects the dynamic patterns of the state accesses, which will tend to decrease cache misses.

6.3.3 Performance

Keeping in mind the difficulties described above, we now present measurements of the performance of the Bro signature engine as compared to Snort. Figure 6.7 shows run-times on trace subsets of different length for the `USB-Full` trace. We show CPU times for the original implementation of Snort, for Snort using [FV01] (virtually no difference in performance), for Snort modified by us as described in the previous section, for Bro with a limited DFA state cache, and for Bro without a limited DFA state cache. We see that our modified Snort runs 18% faster than the original one, while the cache-less Bro takes about the same amount of time. Bro with a limited state cache needs roughly a factor of 2.2 more time.

We might think that the discrepancy between Bro operating with a limited DFA state cache and it operating with unlimited DFA state memory is due to it having to spend considerable time recomputing states previously expired from the limited cache. This, however, turned out not to be the case. Additional experiments with essentially infinite cache sizes indicated that the performance decrease is due to the additional overhead of maintaining the cache.

While this looks like a significant impact, we note that it is not clear whether the space savings of a cache is in fact needed in operational use. For this trace, only 2,669 DFA states had to be computed, totaling roughly 10 MB. When running Bro operationally for a day at the university's gateway, the number of states rapidly climbed to about 2,500 in the first hour, but then from that point on only slowly rose to a bit over 4,000 by the end of the day.

A remaining question, however, is whether an attacker could create traffic specifically tailored to enlarge the DFAs (a "state-holding" attack on the IDS), perhaps by sending a stream of packets that nearly trigger each of the different patterns. Additional research is needed to further evaluate this threat.

Comparing for `USB-Full` the alerts generated by Snort to the signature matches reported by Bro, all in all we find very good agreement. The main difference is the way they report a match. By design, Bro reports all matching signatures, but each one only once per connection. This is similar to the approach suggested in [DM02]. Snort, on the other hand, reports the first matching signature for each packet, independently of the connection it belongs to. This makes it difficult to compare the matches. We account for these difference by comparing connections for which at least one match is generated by either system. With `USB-Full`, we got 2,065 matches by Bro in total on 1,313 connections. Snort reports 4,147 alerts. When counting each alert only once per connection, Snort produced 1,320 on 1,305 connections.[5] There are 1,296 connections for which both generated at least one alert, and 17 (9) for which Bro (Snort) reported a match but not Snort (Bro).

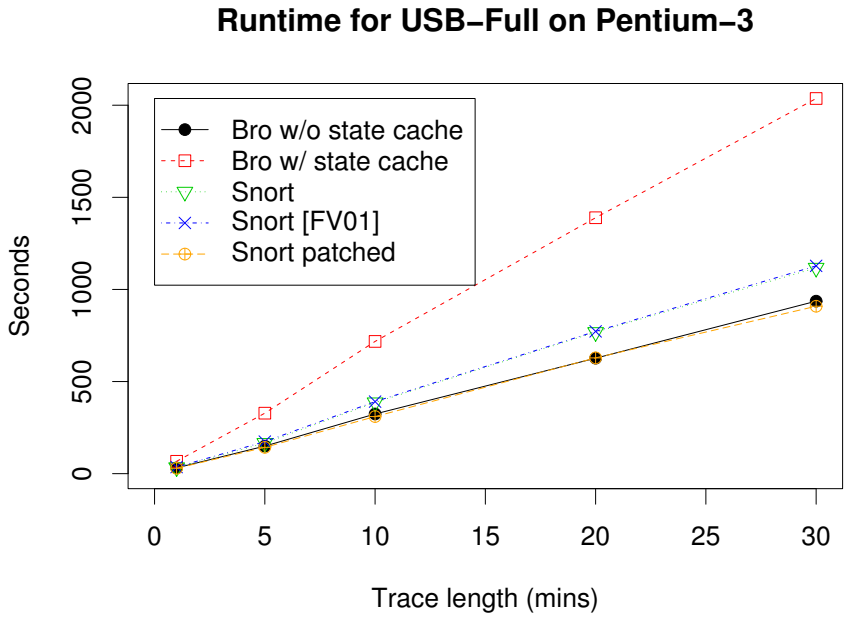
Looking at individual signatures, we saw that Bro missed 10 matches of Snort. 5 of them are caused by Snort ID #1013 (*WEB-IIS fpcount access*). The corresponding connections contained several requests, but an idle time larger than the defined inactivity timeout of 30 seconds. Therefore, Bro flushed the state before it could encounter the match which would had happened later in the session. On the other hand, Bro reported 41 signature matches for connections for which Snort does not report anything. 37 of them are Web signatures. The discrepancy was due to different TCP stream semantics. Bro and Snort have slightly different definitions of when a session is established. Some of the other differences were caused by the semantic differences between stream-wise and packet-wise matching. While Bro's approach allows true stream-wise signatures, it also means that its signature engine loses the notion of "packet size": packets and session payload are decoupled for most of Bro's analyzers. However, Snort's signature format includes a way of specifying the packet size, so when converting such a Snort signature into Bro's format, we must fake up an equivalent by using the size of the first matched payload chunk for each connection. This can lead to differing results.

We did similar measurements with `LBL-Web`. While the original Snort took 440 CPU seconds for the trace, Bro without (with) a limited state cache needed 280 (328) CPU seconds, and Snort as modified by us needed only 164 CPU seconds. While this suggests room for improvement in some of Bro's internal data structures, Bro's matcher still compared quite well to the typical Snort configuration.

For this trace, Bro (Snort) reported 2,764 (2,049) matches in total. If we counted Snort's alerts only once per connection, there were 1,472 of them for 1,464 connections. Bro alerted on 1,528 different connection. There were 1,395 connections for which both reported at least one alert. For 133 (69) connections, Bro (Snort) reported a match but Snort (Bro) did not. Again, looking at individual signatures, Bro missed 73 of Snort's alerts. 25 of them were matches of Snort signature #1287 (*WEB-IIS scripts access*). These were all caused by the same host. The reason were packets missing from the trace, which, due to a lack of in-order sequencing, prevented the TCP stream from being reassembled by Bro. Another 19 are due to signature #1287 (*CodeRed v2 root.exe access*). The ones of these we inspected further were due to premature server-side re-

[5] Most of the duplicates were *ICMP Destination Unreachable* messages.

Figure 6.7 Comparison of processing between Snort and Bro (on 550 MHz Pentium 3).



sets, which Bro correctly identified as the end of the corresponding connections, while Snort kept matching on the traffic still being send by the client. Bro reported 186 signature matches for connections for which Snort did not report a match at all. 68 of these connections simultaneously triggered three signatures (#1002, #1113, #1287). 46 were due to simultaneous matches of signatures #1087 and #1242. Looking at some of them, one reason was SYN-packets missing from the trace. Their absence led to different interpretations of established sessions by Snort and Bro, and therefore to different matches.

6.3.4 Operational Experiences

The first large deployments of contextual signatures occurred within the BroLite project [Broa] at LBNL. The aim of this project—which due to lack of funding is currently on hold—is to develop an end-user friendly version of the Bro NIDS, suitable for deployment by inexperienced users. Due to their conceptual ease, contextual signatures constitute one of the main building-blocks of BroLite: they can be be centrally developed, bundled, and then shipped to users for installation.

The first approach was to leverage Snort's full signature library by means of the converter that we developed (see §6.2.3). As already noted, simply converting traditional signatures does not include context into the matching process though. Therefore, the BroLite group supplemented the converted signatures with additional conditions. For example, in accordance with one of our examples, many HTTP signatures were annotated to match bi-directionally, raising events only if a server indicates a positive response.

Others were extended to take vulnerability profiles into account. As anticipated, annotating the converted signatures needed to be performed manually. This process required considerable efforts, although the BroLite group developed a set of tools to support it.

Given the work that these annotations required, the outcome was not as good as expected. The annotations (in combination with disabling signatures which could not be modified to work reliably) reduced the number of alerts considerably. However, the additional CPU requirements were significant. Bro performs signature matching *in addition* to all of its other work. Therefore, in our high-performance environments, matching thousands of signatures becomes a major burden. Of course, we anticipated this; Snort runs into similar problems in such environments. However, we expected that the benefit of spending considerable CPU time for signature matching would be larger. Unfortunately, even with many annotations in place, the (few) alerts that Bro reported were still mainly false positives, usually from not-yet annotated signatures. We attribute this to the lack of quality that the underlying Snort set exhibits; there are too many old, now irrelevant signatures. While there are certainly also very valuable signatures in the set, we conclude that using the full set of Snort signatures, even if annotated, is not worth the additional CPU demand required in a high-performance environment.

Moreover, an external signature set is a moving target: signature updates need to be incorporated. In the first phase of the BroLite project, the group worked with a frozen set of Snort signatures. Eventually catching up with the latest signatures would have needed another considerable effort. Also, during the work, Snort's signature language evolved, providing more capabilities, some of which we have not yet reproduced in our converter. In particular, Snort also supports regular expressions now, yet uses an incompatible syntax. To support Snort's expressions, we have developed a prototype converter which is able to translate between different styles of regular expressions.

Given these difficulties, in the future we will follow the inverse approach: starting with no signatures at all, we will build up a small set of high-quality signatures, concentrating on detection performance and leveraging all of the context provided by Bro right from the beginning. This approach is in accordance with our observations in §4.2.2: at many sites, Snort is used in this way, ignoring the signature set it ships with.

There are conceptual problems to solve as well. One question involves the separation of policy and mechanism. As discussed in §2.5.2, Bro ships with a set of default policy scripts which the user needs to adapt to the local environment. Similarly, when shipping with a default signature set, the user will need to tune the matching process accordingly. However, we believe it would be beneficial if the user would not need to touch the signatures themselves. If signatures and policy adaptations are separated, signature updates will be significantly easier to perform. (We note that the tools developed by the BroLite project already provision for such a separation.) Another technical problem is Bro's unit of analysis, a connection. For many application-layer protocols, this appears to be too coarse, and leads to problems when correlating context during signature matching. For example, a persistent HTTP connection may contain multiple request/response pairs. Currently, a bi-directional signature is not able to correlate a server response with the *correct* request. This is a major problem which seems to need significant internal changes

of the Bro NIDS to solve in a way that is sufficiently general to apply to other protocols as well. (We note that Bro's *script-level* HTTP analyzer correctly associates requests with replies in persistent connections by using additional script code.)

6.4 Summary

In this chapter, we develop the general notion of *contextual signatures* as an improvement on the traditional form of string-based signature-matching used by NIDS. Rather than matching fixed strings in isolation, contextual signatures augment the matching process with both low-level context, by using regular expressions for matching rather than simply fixed strings, and high-level context, by taking advantage of the rich, additional semantic context made available by Bro's protocol analysis and scripting language.

By tightly integrating the new signature engine into Bro's event-based architecture, we achieve several major improvements over other signature-based NIDSs such as Snort, which frequently suffer from generating a huge number of alerts. By interpreting a signature-match only as an event, rather than as an alert by itself, we are able to leverage Bro's context and state-management mechanisms to improve the quality of alerts. We showed several examples of the power of this approach: matching requests with replies, recognizing exploit scans, making use of vulnerability profiles, and defining dependencies between signatures to model attacks that span multiple connections. In addition, by converting the freely available signature set of Snort into Bro's language, we are able to build upon existing community efforts. Unfortunately, operational experiences supported the often-heard complaint that Snort's signature lack quality. Therefore, for the future we believe it to be more promising to start over with an empty signature set, devising high-quality contextual signatures from scratch.

As a baseline, we evaluated our signature engine using Snort as a reference, comparing the two systems in terms of both run-time performance and generated alerts. But in the process of doing so, we encountered several general problems when comparing NIDSs: differing internal semantics, incompatible tuning options, the difficulty of devising "representative" input, and extreme sensitivity to hardware particulars. The last two are particularly challenging, because there are no *a priori* indications when comparing performance on one particular trace and hardware platform that we might obtain very different results using a different trace or hardware platform. Thus, we must exercise great caution in interpreting comparisons between NIDSs.

7 Conclusion

> There is no single correct level of security; how much security you have depends on what you're willing to give up in order to get it. This trade-off is, by its very nature, subjective—security decisions are based on personal judgments.
>
> — *Bruce Schneier, "Beyond Fear"*

7.1 Summary

High-performance networks pose major challenges for network intrusion detection. A common experience when installing an arbitrary NIDS in such an environment is that, in its out-of-the-box configuration, it essentially fails to work: a typical NIDS generates lots of false alerts, and often does not have the required processing performance to keep up with the packet stream. The process of tuning such a system to the local environment is time-consuming and requires significant expertise. However, even carefully tuned systems often do not meet the ambitious promises of their vendors.

In this work, we set out to bridge the gap between claims and operational reality in network intrusion detection. Our initial motivation was the experience of setting up the open-source Bro NIDS in a medium-scale environment. While the Bro system provided a great deal of power and flexibility, we noticed that it lacked the mechanisms required to tune its resource demands to the specifics of our network. We started to analyze the concepts of NIDSs in order to understand the trade-offs and limitations that such systems face.

Overall, we found that there are two main trade-offs involved in high-performance network intrusion detection: detection quality versus resource demands, and detection quality versus false positive rate. Having identified these trade-offs, we developed concepts and mechanisms to mitigate their impact. Our study is based on our experience with the network traffic of several large network environments.

Traffic in High-Performance Environments

One of the major observations about high-performance environments is that they exhibit an immense traffic diversity. In the microscopic per-packet view of a NIDS, traffic characteristics are essentially unpredictable. To work reliably, a NIDS needs to deal robustly with different application mixes, handle large fluctuations in traffic characteristics, and cope with unexpected content. If a NIDS is built just for the "average case", it will frequently fail.

The traffic's application mix has a major impact on a NIDS's performance as these systems analyze different types of applications to different depths. An environment's mix is determined by the availability and popularity of services; a super-computing environment shows a different application mix than a university. While overall, for a given environment, the mix tends to be relatively stable, on a daily basis it can fluctuate considerably due to time-of-day effects and extreme situations such as down-times of major servers, flash crowds, attacks, or widely misbehaving software.

On smaller time-scales, i.e., minutes/seconds, the "burstiness" of network traffic requires a NIDS to keep large safety-margins. Frequently, there are large peaks in, e.g., traffic volume, session number and session duration, which the system needs to be prepared for.

Robustness is also required when analyzing traffic content. An attacker may deliberately craft packets not conforming to protocol specifications. Even without any adversary, real-world traffic is so diverse that, given a sufficiently large traffic stream, many protocol corner-cases will eventually manifest. We regularly encounter blatant protocol violations even in benign traffic. Thus, a NIDS needs to expect unexpected traffic content.

While, in principle, such effects exist in all networks, in high-performance networks their impact is crucial. A high-volume packet stream, composed of the communications of thousands of hosts, amplifies any difficulty.

Detection Quality versus Resource Demands

As noted above, to deal robustly with traffic of a high-performance environment, it is not sufficient for a NIDS to only support "average" situations. However, designing for the *worst* case is not feasible either: we would need an unlimited supply of CPU cycles and memory to accommodate any conceivable situation. Thus, it is necessary to find a point in between. Essentially one needs to trade-off the precision of the NIDS's detection with the resources, in terms of CPU and memory, that can be supplied.

Finding such a trade-off requires defining priorities. Does one prefer to analyze all of the traffic skin-deep, or to limit the analysis to a fraction of the traffic yet examine that part thoroughly? Is it acceptable to potentially miss evasion attacks when this allows to include more traffic into the overall analysis? Which kinds of traffic spikes are supposed to be handled, i.e., what is the required safety-margin?

Answers to such questions are environment-specific. Therefore, a NIDS needs to supply the mechanisms required to adapt the systems to the answers. Based on our experience with operational networks, we devised a toolbox of such mechanisms which provides two orthogonal, and hence combinable, approaches: first, it provides means to parametrize a NIDS's resource and load management in a very flexible way, thereby enabling detailed tuning to environment specifics (see §4.5); second, it allows to multiply the available resources by transparently sharing state across multiple instances of the system, thereby allowing to choose a more conservative trade-off (see §5.5.3).

Detection Quality versus False Positive Rate

A NIDS can rarely be perfectly sure that some activity indeed constitutes an attack. Often, a benign situation is also conceivable in which the very same activity can be observed. If the NIDS alerts on anything which could potentially be malicious, it will generate an unmanageable number of false alerts. If, on the other hand, it only reports activity which, with a very high probability, is a real intrusion, it will miss some. Similar to the quality/resource trade-off, it is an environment-specific decision what degree of "suspectedness" warrants an alert.

In high-performance environments, the traffic volume again amplifies the effect of the chosen trade-off. If one requires to see *all* attacks, there will be so many false positives, that the correct alerts among them can hardly be identified. On the other hand, if one tunes the system to hide all but the most obvious attacks, one will indeed miss a considerable number of intrusions.

Our approach to mitigate this effect is to improve the precision of the detection by incorporating context into the analysis. An increased reliability of alerts allows the operator to also increase the level of required "suspectedness" without running the risk of missing significantly more attacks than before.

The key observation here is that almost all false alerts are due the NIDS lacking some "background information". If benign activity appears to be malicious to the NIDS, the system lacks the knowledge to determine that the situation in fact represents valid use. For example, a local host may be connecting to a large number of apparently random Web servers in a very short time interval. This could be due to someone scanning the Internet for vulnerable Web servers—which would probably warrant an alert. However, if one knows that the host is in fact a large HTTP proxy, the observed activity is certainly within its expected behavior.

Thus, we strive to increase the amount of context available in the detection process. While most NIDS already make use of contextual information (e.g., by remembering connection state, tracking host behavior, or incorporating knowledge from external databases), we enhance the use of context in three different ways:

1. We ensure that once-available context is not lost unexpectedly, i.e., we make it persistent (see §5.5.1). Usually, much of the fine-grained internal state of a NIDS is lost upon (regular or unexpected) termination. That is, all of the contextual information that the system has derived is discarded at this point. By keeping state across restarts, we enable the NIDS to constantly refine its picture of the network without starting from scratch all the time.

2. We allow context to be shared among instances of the NIDS as well as to be provided by external applications. Sharing state allows a NIDS to leverage context which it is not able to derive itself (see §5.5.3). Including external information loosens the NIDS restriction to purely network-based analysis; it may now, e.g., include host-based context (see §5.7).

3. We provide mechanisms to efficiently *use* the context accumulated by a NIDS. In most NIDS, byte-wise matching of known attack patterns is still the prevalent de-

tection method. While highly efficient, this approach, by itself, completely ignores any available context. Our approach considers the match of such a pattern only as the starting point of a process which then uses context to support the decision (see §6.2).

Of course, collecting, storing, and using contextual information requires resources. Thus, leveraging context is also another aspect of the quality/resource trade-off. For example, in a high-performance network, it is neither practical nor preferable to make *all* available context persistent. However, our tuning toolbox also provides the mechanisms to adapt the use of contextual information as required (see §5.4.2).

A High-Performance NIDS

We implemented all of our extensions within the framework provided by the open-source Bro NIDS. While at the beginning of our work, we had trouble running the system in a medium-scale network, it is now suitable for use in all of the high-performance environments that we have examined. The implementations of our concepts of *independent state* and *contextual signatures* greatly extend the system's power. Our toolbox of tuning options provides the mechanisms to adapt the system to the networks' different characteristics. All of our extensions are now included in the Bro distribution.

7.2 Outlook

Our work provides us with an in-depth understanding of the issues involved in high-performance network intrusion detection, as well as with an open-source NIDS suitable for use in environments of all scales. There are several promising research directions open for future work; among them are widely distributed setups, improved usability as well as identifying the most powerful unit of analysis.

Widely Distributed Network Intrusion Detection

One of the most promising areas for future work is widely distributed network intrusion detection (and prevention). Today, most attacks against information systems do not originate from intruders nearby. In fact, most attacks are not even directed at individual systems but compromise whatever they find to be vulnerable.

Despite such a global threat, detection is still pursued almost solely locally. While many network intrusion detection systems are able to operate multiple sensors inside an organization's network, their correlation techniques are still quite crude. When looking at Internet-scale detection, currently we only find a few rather half-hearted data collection initiatives (e.g., DShield [DSh]) which lack the quality required for reliable results.

We believe that an Internet-scale distribution of cooperating detection components will significantly increase the ability to identify and counter attacks, both undirected and directed. A prominent example of large undirected attacks is worms which, due to

their potentially immense speed, pose a very real threat to the Internet. Distributing many worm sensors across the Internet promises to quickly detect new outbreaks, which is a requirement for any effective counter-measure.

An example of large-scale directed attacks is distributed denial-of-service floods. During such a flood tens to hundreds of thousands of hosts ("zombies" which have been compromised themselves) overwhelm the victim by sending bogus data. Despite a significant body of research, in practice there is still no effective way to quickly escape such a flood once it is in progress (besides moving the victim away). We believe that it is necessary to intensify the efforts for identifying zombies *before* they start to attack. A coordinated deployment of "zombie-detectors" across the Internet should be able to find a large number of such hosts.

However, coordinating a distributed detection setup is challenging. Even at medium scales, e.g., within a single organization, this task is non-trivial as different departments tend to have different demands and policies. At Internet-scale, it gets even more difficult as independently organized networks are involved. To cooperate, they need to communicate. This is by itself a hard problem. Apart from technical questions, major political and social issues arise. For example, no network can provide anybody access to its full traffic; even if an organization itself would not object, privacy laws do not permit it. Therefore, one needs to find ways to separate sensitive data from non-sensitive data (for instance by anonymizing). More generally, models for formulating data sharing policies need to be developed. As a step in this direction, [ABP05] sketches an architecture for sharing past behavioral patterns about, e.g., network hosts.

This leads to the question for which kinds of data it makes most sense to share them, given organizational restrictions (e.g., privacy requirements) and technical limitations (e.g., data volume). One candidate is connection-level information: while one would not share any connection content, propagating which hosts have communicated can be a major help in detecting attacks like worms. However, IP-addresses may also be considered sensitive.

Developing a comprehensive model of cooperation for distributed detection looks intriguing. We have already extended the Bro NIDS to apply its event-based approach to distributed environments. We believe that in this context a model can be developed which captures the system's policy-neutral characteristics, enabling the use of locally specified cooperation policies.

Usability of Network Intrusion Detection Systems

It is a very challenging task to design and implement a user interface for a NIDS that is effective to use for users with different levels of expertise. None of the user interfaces that we have seen so far provides the tools that we deem necessary to exploit the full potential of network intrusion detection.

For analyzing the output of a NIDS, we envision a user interface based on two primary concepts: correlation and interactivity. Most current (graphical) user interfaces provide means to group related alerts based on common attributes. However, there are rarely any

more sophisticated correlation schemes in use which exploit non-obvious relationships. In fact, the research on correlation is still at a rather immature state, leaving the right approach unclear.

In any case, we believe that correlation is primarily a tool to *support* the user when examining a NIDS's output, rather than a fully automatic mechanism to find relevant alerts. Therefore, a user interface needs to integrate correlation techniques into a highly-interactive interface. The user needs to be able to quickly jump back and forth between different levels of abstractions as well as different kinds of correlations. Moreover, relevant contextual information needs to be easily accessible (e.g., a summary of a host's activity). A high responsiveness is of utmost importance for such a degree of interactivity. A user should be able to "play" with the data to explore it many different ways.

Unit of Analysis

In several different contexts we noticed that a connection may not be the optimal unit of analysis for a NIDS. While packets are clearly too low-level, it appears that there is a unit in between which would provide the most power.

Consider for example a persistent HTTP connection which consists of several independent HTTP request/reply pairs. In this case the right unit appears to be such a pair instead of the surrounding connection, which is primarily a mechanism to improve performance rather than having any semantic meaning other than defining the involved hosts. Another example is the SSH protocol which multiplexes multiple independent channels within a single connection. For this protocol, such a channel seems to be an appropriate unit. Yet, channels contain activity for which further refinement makes sense (e.g., interactively entered command-lines and their server-side results).

Also due to the connections being the units of analysis, we encountered difficulties when we interfaced external host applications with the Bro NIDS (see §5.7): for externally provided information, the NIDS, in general, does not have any associated connection state. Yet, most most of the system's analysis requires such state for its operations.

It appears that a generic notion of *transactions* being the unit of analysis is suitable to solve such problems. Using an object-oriented scheme, analyzers can define different kinds of transactions, specializing them with analyzer-specific state. Moreover, transactions can be nested, containing other transactions in turn. In the HTTP example, the request/reply pairs form a list of transactions. In the SSH example, each channel is a transaction, potentially containing more transactions of a different kind itself. External sensors can send transactions containing just the information required for their analysis.

Bibliography

[ABP05] Mark Allman, Ethan Blanton, and Vern Paxson. An Architecture for Developing Behavioral History. In *Proc. Workshop on Steps to Reducing Unwanted Traffic on the Internet*, 2005.

[ACI] ACID. `http://acidlab.sourceforge.ne`.

[AGJT03] Deb Agarwal, Jose Maria Gonzalez, Goujun Jin, and Brian Tierney. An infrastructure for Passive Network Monitoring of Application Data Streams. In *Proc. Passive and Active Measurement Workshop*, 2003.

[AL01] Magnus Almgren and Ulf Lindqvist. Application-Integrated Data Collection for Security Monitoring. In *Proc. of Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[Axe99a] Stefan Axelsson. Intrusion Detection Systems: A Survey. Technical Report 98-17, Chalmers University of Technology, Sweden, 1999.

[Axe99b] Stefan Axelsson. The Base-Rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In *Proc. ACM Conference on Computer and Communications Security*, 1999.

[Axe00] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security*, 3(3):186–205, August 2000.

[Bac00] Rebecca Gurley Bace. *Intrusion Detection*. Macmillan Technical Publishing, 2000. ISBN 1-578-70185-6.

[BF00] Joachim Biskup and Ulrich Flegel. Threshold-based Identity Recovery for Privacy Enhanced Applications. In *Proc. ACM Conference on Computer and Communications Security*, 2000.

[BFIM98] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2616, 1998.

[Bis03] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley, 2003. ISBN 0-201-44099-7.

[Bla] CERT Advisory CA-2003-20 W32/Blaster worm. `http://www.cert.org/advisories/CA-2003-20.html`.

[Bld] BleedingSnort. `http://www.bleedingsnort.com`.

[Broa] Bro Intrusion Detection System. `http://www.bro-ids.org`.

[Brob] Broccoli: The Bro Client Communications Library. `http://www.cl.cam.ac.uk/~cpk25/broccoli`.

[Bug] SecurityFocus Vulnerability Database. `http://www.securityfocus.com/bid`.

[CA] CERT Advisory CA-2002-27 Apache/mod_ssl Worm. `http://www.cert.org/advisories/CA-2002-27.html`.

[CBR03] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker, Second Edition*. Addison-Wesley, 2003. ISBN 0-201-63466-X.

[Cis] Cisco Netflow. `http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml`.

[CM02] Frédéric Cuppens and Alexandre Miège. Alert Correlation in a Cooperative Intrusion Detection Framework. In *Proc. IEEE Symposium on Security and Privacy*, 2002.

[CO00] Frédéric Cuppens and Rodolphe Ortalo. LAMBDA: A Language to Model a Database for Detection of Attacks. In *Proc. Recent Advances in Intrusion Detection*, number 1907 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[CSM01] C. Jason Coit, Stuart Staniford, and Joseph McAlerney. Towards Faster Pattern Matching for Intrusion Detection or Exceeding the Speed of Snort. In *Proc. DARPA Information Survivability Conference and Exposition*, 2001.

[CVE] Common Vulnerabilities and Exposures. `http://www.cve.mitre.org`.

[CVE01a] CVE-2001-0333. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0333`, 2001.

[CVE01b] CVE-2002-0601. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0601`, 2001.

[CVE02a] CVE-2002-0115. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0115`, 2002.

[CVE02b] CVE-2001-0144. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144`, 2002.

[CW03] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proc. USENIX Security Symposium*, 2003.

[DC] The California K-12 High Speed Network. `http://www.cenic.org/dcp/index.htm`.

[Den87] Dorothy E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[Der03] Luca Deri. Improving Passive Packet Capture: Beyond Device Polling. Technical report, University of Pisa, 2003.

[Der04] Luca Deri. Improving Passive Packet Capture:Beyond Device Polling. In *Proc. International System Administration and Network Engineering Conference*, 2004.

[DM02] Hervé Debar and Benjamin Morin. Evaluation of the Diagnostic Capabilities of Commercial Intrusion Detection Systems. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[DP05] Sarang Dharmapurikar and Vern Paxson. Robust TCP Reassembly in the Presence of Adversaries. In *Proc. USENIX Security Symposium*, 2005.

[Dra] Enterasys Intrusion Defense. `http://www.enterasys.com/products/ids`.

[DSh] Distributed Intrusion Detection System *DShield.org*. `http://www.dshield.org`.

[DW01] Hervé Debar and Andreas Wespi. Aggregation and Correlation of Intrusion-Detection Alerts. In *Proc. Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[End] ENDACE Measurement Systems. `http://www.endace.com`.

[EVK02] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2): 71–104, 2002.

[FFM04] Michael J. Freedman, Eric Freudenthal, and David Maziymposium. Democratizing Content Publication with Coral. In *Proc. Symposium on Networked Systems Design and Implementation*, 2004.

[FGB+03] Wu-chang Feng, Ashvin Goel, Abdelmajid Bezzaz, Wu-chi Feng, and Jonathan Walpole. TCPivo: A High Performance Packet Replay Engine. In *Proc. Workshop on Models, Methods, and Tools for Reproducible Network Research*, 2003.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, 1999.

[FGW98] Anja Feldmann, Anna C. Gilbert, and Walter Willinger. Data Networks As Cascades: Investigating the Multifractal Nature of Internet WAN Traffic. In *Proc. ACM SIGCOMM*, 1998.

[FV01] Mike Fisk and George Varghese. Fast Content-Based Packet Handling for Intrusion Detection. Technical Report CS2001-0670, University of Calfornia, San Diego, 2001.

[Fyo98] Fyodor. Remote OS Detection via TCP/IP Stack Finger Printing. *Phrack Magazine*, 8(54), 1998.

[Gon05] Jose Maria Gonzalez. *Efficient Filtering Support for High-Speed Network Intrusion Detection*. PhD thesis, University of California, Berkeley, 2005.

[GPL] GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`.

[GPr] GNU Binutils. `http://www.gnu.org/software/binutils`.

[HH04] Simon Hansman and Ray Hunt. A Classification of Attack Methodologies for Use by CERTs. In *Proc. Asia-Pacific Region Internet Conference of Technology*, 2004.

[HKP01] Mark Handley, Christian Kreibich, and Vern Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proc. USENIX Security Symposium*, 2001.

[HKR92] J. Heering, P. Klint, and J. Rekers. Incremental Generation of Lexical Scanners. *ACM Transactions on Programming Languages and Systems*, 14(4): 490–520, 1992. ISSN 0164-0925.

[HM04] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Addison Wesley, 2004. ISBN 0-201-78695-8.

[Hof99] Steven Andrew Hofmeyr. *An Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, University of New Mexico, 1999.

[HPP] Mass Storage at NERSC. `http://www.nersc.gov/nusers/resources/HPSS`.

[HPR] CENIC Network. `http://www.cenic.org/calren/index.htm`.

[HRLC01] Joshua Haines, Lee Rossey, Richard Lippmann, and Robert Cunnigham. Extending the 1999 Evaluation. In *Proc. DARPA Information Survivability Conference and Exposition*, June 2001.

[HS01] James A. Hoagland and Stuart Staniford. Viewing IDS Alerts: Lessons from SnortSnarf. In *Proc. DARPA Information Survivability Conference and Exposition*, 2001.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979. ISBN 0-201-02988-X.

[HW02] Mike Hall and Kevin Wiley. Capacity Verification for High Speed Network Intrusion Detection Systems. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[ICS] International Computer Science Institute. `http://www.icsi.berkeley.edu`.

[IDM] Intrusion Detection Message Exchange Format. `http://www.ietf.org/html.charters/idwg-charter.html`.

[Ilg93] Koral Ilgun. USTAT: A Real-Time Intrusion Detection System for UNIX. In *Proc. IEEE Symposium on Security and Privacy*, 1993.

[Int] McAfee IntruShield Network IPS Appliances. `http://www.networkassociates.com/us/products/mcafee/network_ips/intrush%ield_appliances.htm`.

[Int01] Internet Security Systems Security Alert. Multiple Vendor IDS Unicode Bypass Vulnerability. `http://xforce.iss.net/xforce/alerts/id/advise95`, 2001.

[Jac99] Kathleen Jackson. Intrusion Detection System Product Survey. Technical Report LA-UR-99-3883, Los Alamos National Laboratory, 1999.

[JCdCM98] José Maurício Bonifácio Jr, Andriano M. Cansian, André C.P.L.F. de Carvalho, and Edson S. Moreira. Neural Networks Applied in Intrusion Detection Systems. In *Proc. IEEE Neural Networks*, 1998.

[Joh01] Steven R. Johnston. The Impact of Privacy and Data Protection Legislation on the Sharing of Intrusion Detection Information . In *Proc. Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[Joh02] Steven R. Johnston. Development of a Legal Framework for Intrusion Detection. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[JPBB04] Jaeyeon Jung, Vern Paxson, Arthur W Berger, and Hari Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proc. IEEE Symposium on Security and Privacy*, 2004.

[Jul03] Klaus Julisch. Clustering Intrusion Detection Alarms to Support Root Cause Analysis. *ACM Transactions on Information and System Security*, 6(4):443–471, 2003.

[JV93] Harold S. Javitz and Alfonso Valdes. The NIDES Statistical Component: Description and Justification. Technical report, SRI International, 1993.

[KFL94] Calvin Ko, George Fink, and Karl Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. In *Proc. Annual Computer Security Applications Conference*, 1994.

[KMVV03] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the Detection of Anomalous System Call Arguments. In *Proc. European Symposium on Research in Computer Security*, 2003.

[KO04] Hideki Koike and Kazuhiro Ohno. SnortView: Visualization System of Snort Logs. In *Proc. ACM Workshop on Visualization and Data Mining for Computer Security*, 2004.

[KPD+05] Stefan Kornexl, Vern Paxson, Holger Dreger, Anja Feldmann, and Robin Sommer. Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic. In *Proc. Internet Measurement Conference*, 2005.

[KRL97] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proc. IEEE Symposium on Security and Privacy*, 1997.

[KS94] Sandeep Kumar and Eugene H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proc. National Computer Security Conference*, 1994.

[KS05] Christian Kreibich and Robin Sommer. Policy-Controlled Event Management for Distributed Intrusion Detection. In *Proc. International Workshop on Distributed Event-Based Systems*, 2005.

[KV03] Christopher Kruegel and Giovanni Vigna. Anomaly Detection of Web-based Attacks. In *Proc. ACM Conference on Computer and Communications Security*, 2003.

[LBL] Lawrence Berkeley National Laboratory. `http://www.lbl.gov`.

[LCT+02] Wenke Lee, Joao B.D. Cabrera, Ashley Thomas, Niranjan Balwalli, Sunmeet Saluja, and Yi Zhang. Performance Adaptation in Real-Time Intrusion Detection Systems. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[LFG+00] Richard P. Lippmann, David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David McClung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunningham, and Marc A. Zissman. Evaluating Intrusion Detection Systems: the 1998 DARPA Off-Line Intrusion Detection Evaluation. In *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition*, 2000.

[LHF+00] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, 2000.

[Lib] libpcap. `http://www.tcpdump.org`.

[LJ97] Ulf Lindqvist and Erland Jonsson. How to Systematically Classify Computer Security Intrusions. In *Proc. IEEE Symposium on Security and Privacy*, 1997.

[LP99] Ulf Lindqvist and Phillip A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In *Proc. IEEE Symposium on Security and Privacy*, 1999.

[LPV04] Kirill Levchenko, Ramamohan Paturi, and George Varghese. On the Difficulty of Scalably Detecting Network Attacks. In *Proc. ACM Conference on Computer and Communications Security*, 2004.

[LW] libwhisker. `http://www.wiretrip.net/rfp`.

[LWS02] Richard Lippmann, Seth Webster, and Douglas Stetson. The Effect of Identifying Vulnerabilities and Patching Software on the Utility of Network Intrusion Detection. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[LX01] Wenke Lee and Dong Xiang. Information-Theoretic Measures for Anomaly Detection. In *Proc. IEEE Symposium on Security and Privacy*, 2001.

[McH00] John McHugh. Testing Intrusion detection systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory. *ACM Transactions on Information and System Security*, 3(4):262–294, 2000.

[MD03] Benjamin Morin and Hervé Debar. Correlation of Intrusion Symptoms: An Application of Chronicles. In *Proc. Recent Advances in Intrusion Detection*, number 2820 in Lecture Notes in Computer Science. Springer-Verlag, 2003.

[MDDR05] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. *Internet Denial of Service: Attack and Defense Mechanisms.* Prentice Hall, 2005. ISBN 0-131-47573-8.

[MIT] The MITRE Corporation. `http://www.mitre.org`.

[MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. Winter USENIX Conference*, 1993.

[MJ98] D. Mosberger and T. Jin. httperf - A Tool For Measuring Web Server Performance. In *Proc. Workshop on Internet Server Performance*, 1998.

[ML] Posting to the focus-ids mailing list. `http://seclists.org/lists/focus-ids/2005/Apr/0052.html`.

[MMDD02] Benjamin Morin, Ludovic Mé, Hervé Debar, and Mireille Ducassé. M2D2: A Formal Data Model for IDS Alert Correlation. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[MMP] A Libpcap Version which Supports MMAP Mode on the Linux Kernel. `http://public.lanl.gov/cpw`.

[Mod] mod_benchmark Apache plugin. `http://www.trickytools.com/php/mod_benchmark.php`.

[MPa] `mpatrol`. `http://www.cbmamiga.demon.co.uk/mpatrol`.

[MPS+03] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. *IEEE Magazine of Security and Privacy*, 2003.

[MS04] David Moore and Colleen Shannon. The Spread of the Witty Worm. `http://www.caida.org/analysis/security/witty`, 2004.

[MVS01] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet Denial-of-Service Activity. In *Proc. USENIX Security Symposium*, 2001.

[MWNa] Das Münchner Wissenschaftsnetz (MWN). `http://www.lrz-muenchen.de/services/netz/mwn-netzkonzept/mwn-netzkonzep%t.pdf`.

[MWNb] Überlick über das Münchner Wissenschaftsnetz. `http://www.lrz-muenchen.de/services/netz/mhn-ueberblick`.

[MWNc] Beschränkungen und Monitoring im Münchner Wissenschaftsnetz. `http://www.lrz-muenchen.de/services/netz/einschraenkung`.

[MWNd] Benutzungsrichtlinien für Informationsverarbeitungssysteme des Leibniz-Rechenzentrums der Bayerischen Akademie der Wissenschaften. `http://www.lrz-muenchen.de/wir/regelwerk/benutzungsrichtlinien`.

[NCR02] Peng Ning, Yun Cui, and Douglas S. Reeves. Constructing Attack Scenarios through Intrusion Alerts. In *Proc. ACM Conference on Computer and Communications Security*, 2002.

[NER] National Energy Research Scientific Computing Center. `http://www.nersc.gov`.

[NET] Network Architecture Group, TU München. `http://www.net.in.tum.de`.

[Nik] Nikto. `http://www.cirt.net/code/nikto.shtml`.

[NKS05] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proc. IEEE Symposium on Security and Privacy*, 2005.

[NP89] Peter G. Neumann and Donn B. Parker. A Summary of Computer Misuse Techniques. In *Proc. National Computer Security Conference*, 1989.

[NSS] The NSS Group. `http://www.nss.co.uk`.

[Ope] `OpenSSL`. `http://www.openssl.org`.

[P0F] Passive OS Fingerprinting. `http://lcamtuf.coredump.cx/p0f.shtml`.

[Pax99] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[PD01] Jean-Phillippe Pouzol and Mireille Ducassé. From Declarative Signatures to Misuse IDS. In *Proc. Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[PF95] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–224, 1995.

[PFV02] Phillip A. Porras, Martin W. Fong, and Alfonso Valdes. A Mission-Impact-Based Approach to INFOSEC Alarm Correlation. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[PLS04] Phillip Porras Patrick Lincoln and Vitaly Shmatikov. Privacy-Preserving Sharing and Correlation of Security Alerts. In *Proc. USENIX Security Symposium*, 2004.

[PN97] Phillip A. Porras and Peter G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *National Information Systems Security Conference*, 1997.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[Pos81] John Postel. Internet Protocol. RFC 791, 1981.

[PP03] Ruoming Pang and Vern Paxson. A High-Level Programming Environment for Packet Trace Anonymization and Transformation. In *Proc. ACM SIGCOMM*, 2003.

[Pup99] Rain Forrest Puppy. A Look At Whisker's Anti-IDS Tactics. `http://www.wiretrip.net/rfp/pages/whitepapers/whiskerids.html`, 1999.

[PV98] Phillip A. Porras and Alfonso Valdes. Live traffic analysis of TCP/IP gateways. In *Proc. ISOC Symposium on Network and Distributed System Security*, 1998.

[PYB+04] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet Background Radiation. In *Proc. Internet Measurement Conference*, 2004.

[QL03] Xinzhou Qin and Wenke Lee. Statistical Causality Analysis of INFOSEC Alert Data. In *Proc. Recent Advances in Intrusion Detection*, number 2820 in Lecture Notes in Computer Science. Springer-Verlag, 2003.

[Ran01] Marcus J. Ranum. Experiences Benchmarking Intrusion Detection Systems. Technical report, NFR Security, Inc., 2001.

[RDFS04] Andy Rupp, Holger Dreger, Anja Feldmann, and Robin Sommer. Packet Trace Manipulation Framework for Test Labs. In *Proc. Internet Measurement Conference*, 2004.

[Roe] Daniel J. Roelker. URL encoder. `http://code.idsresearch.org/encoder.c`.

[Roe99] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. Systems Administration Conference*, 1999.

[Roe04] Daniel J. Roelker. HTTP IDS Evasions Revisited. `http://www.sourcefire.com/products/downloads/secured/sf_HTTP_IDS_evasio%ns.pdf`, 2004.

[RSP] Configuring SPAN and RSPAN (Cisco Catalyst 6500 Series). `http://www.cisco.com/univercd/cc/td/doc/product/lan/cat6000/sw_7_5/conf%g_gd/span.pdf`.

[RSS99] S. Verma R. Sekar, Y. Guang and T. Shanbhag. A High-Performance Network Intrusion Detection System. In *Proc. ACM Conference on Computer and Communications Security*, 1999.

[SBD+91] Steven R. Snapp, James Brentano, Gihan V. Dias, Terrance L. Goan, L. Todd Heberlein, Che-Lin Ho, Karl N. Levitt, Biswanath Mukherjee, Stephen E. Smaha, Tim Grance, Daniel M. Teal, and Doug Mansur. DIDS (Distributed Intrusion Detection System) – Motivation, Architecture, and an Early Prototype. In *Proc. NIST National Computer Security Conference*, 1991.

[SCC+96] 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 *Proc. NIST-NCSC National Information Systems Security Conference*, 1996.

[SF02] Robin Sommer and Anja Feldmann. NetFlow: Information Loss or Win? In *Proc. ACM SIGCOMM Internet Measurement Workshop*, 2002.

[SGF+02] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-Based Anomaly Detection: a New Approach for Detecting Network Intrusions. In *Proc. ACM Conference on Computer and Communications Security*, 2002.

[SHM02] Stuart Staniford, James A. Hoagland, and Joseph M. McAlerney. Practical Automated Detection of Stealthy Portscans. *Journal of Computer Security*, 10(1-2):105–136, 2002.

[SMPW04] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The Top Speed of Flash Worms. In *Proc. ACM Workshop on Rapid Malcode*, 2004.

[Sno] Snot. `http://www.stolenshoes.net/sniph/index.html`.

[Sou94] Jiri Soukup. *Taming C++ – Pattern Classes and Persistence for Large Projects*. Addison-Wesley, 1994. ISBN 0-201-52826-6.

[SP03] Umesh Shankar and Vern Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *Proc. IEEE Symposium on Security and Privacy*, 2003.

[SPM99] Chris Sinclair, Lyn Pierce, and Sara Matzner. An Application of Machine Learning to Network Intrusion Detection. In *Proc. Computer Security Applications Conference*, 1999.

[SPW02] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in Your Spare Time. In *Proc. USENIX Security Symposium*, 2002.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley Professional, 1994. ISBN 0-201-63346-9.

[Sti] Stick. `http://packetstormsecurity.nl/distributed/stick.htm`.

[Sti03] Richard Stiennon. Intrusion Detection is Dead - Long Live Intrusion Prevention. Technical report, Gartner Inc., 2003.

[SW05] Fabian Schneider and Jörg Wallerich. Performance Evaluation of Packet Capturing Systems for High-Speed Networks. In *Proc. CoNEXT*, 2005.

[TCP] `tcpdump`. `http://www.tcpdump.org`.

[TM] Kymie M.C. Tan and Roy A. Maxion. "Why 6?" Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector.

[TM02] Kymie M.C. Tan and Kevin S. Killourhyand Roy A. Maxion. Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[TS02] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems – Principles and Paradigms*. Prentice Hall, 2002. ISBN 0-13-088893-1.

[UCB] University of California, Berkeley. `http://www.berkeley.edu`.

[USB] Universität des Saarlandes. `http://www.uni-saarland.de`.

[Val] Valgrind. `http://developer.kde.org/~sewardj`.

[VEK00] Giovanni Vigna, Steven T. Eckmann, and Richard A. Kemmerer. The STAT Tool Suite. In *Proc. DARPA Information Survivability Conference and Exposition*, 2000.

[VK99] Giovanni Vigna and Richard A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.

[VKB01] Giovanni Vigna, Richard A. Kemmerer, and Per Blix. Designing a Web of Highly-Configurable Intrusion Detection Sensors. In *Proc. of Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science, 2001.

[VS01] Alfonso Valdes and Keith Skinner. Probilistic Alert Correlations. In *Proc. Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[WH01] Marc Welz and Andrew Hutchison. Interfacing Trusted Applications with Intrusion Detection Systems. In *Proc. of Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[Whi] Whisker. `http://www.wiretrip.net/rfp`.

[WPSC03] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A Taxonomy of Computer Worms. In *Proc. ACM Computer and Communications Security Workshop on Rapid Malcode*, 2003.

[WS02] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proc. ACM Conference on Computer and Communications Security*, 2002.

[WTSW97] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level. *IEEE/ACM Transactions on Networking*, 5(1), 1997.

[ZLM+01] Zheng Zhang, Jun Li, C.N. Manikopoulos, Jay Jorgenson, and Jose Ucles. HIDE: a Hierarchical Network Intrusion Detection System Using Statistical Preprocessing and Neural Network Classification. In *Proc. IEEE Workshop on Information Assurance and Security*, 2001.

[ZP00a] Yin Zhang and Vern Paxson. Detecting Backdoors. In *Proc. USENIX Security Symposium*, 2000.

[ZP00b] Yin Zhang and Vern Paxson. Detecting Stepping Stones. In *Proc. USENIX Security Symposium*, 2000.