Lehrstuhl für Netzwerkarchitekturen
Fakultät für Informatik
Technische Universität München

Operational Network Intrusion Detection: Resource-Analysis Tradeoffs

Holger Dreger

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. Dr. Helmut Veith
Prüfer der Dissertation:	1.	Univ.-Prof. Anja Feldmann, Ph.D. Technische Universität Berlin
	2.	Univ.-Prof. (komm. L.) Dr. Thomas Fuhrmann
	3.	Prof. Vern Paxson, Ph.D. University of California, Berkeley / USA

Die Dissertation wurde am 07.03.2007 bei der Technischen Universität München
eingereicht und durch die Fakultät für Informatik am 03.07.2007 angenommen.

Abstract

Network Intrusion Detection Systems (NIDS) span an area of massive research and commercial interest. Modern systems offer a wide range of capabilities and parameters to adapt the analysis to the needs of the operator. If one deploys a NIDS in a high volume network environment (1Gbps or more) however, one notices that some capabilities are not usable as the available resources (CPU cycles and memory) of the NIDS are not sufficient for such detailed analysis.

In this thesis, we target a thorough understanding of the dependencies and tradeoffs between NIDS resource usage and detection capabilities. We base this work on our operational experience with NIDS in three large research network environments, among them the Münchener Wissenschaftsnetz (*MWN*), Germany. We demonstrate, that operational network intrusion detection in high-volume network environments raises a host of resource management issues. We explore tradeoffs between resource usage and analysis, that range from predictive to adaptive to retrospective.

Predicting the resource usage of a NIDS is difficult. We set out to develop a performance model of a NIDS that allows to determine the appropriate analysis depth and parametrization of the system. The model can be used in two ways: First, to help determining a configuration of the NIDS based on the measured traffic characteristics of the network environment. Second, it can be used to predict the NIDS' resource usage based on a known or guessed trend of the network traffic development.

Connection oriented NIDS do not analyze every connection the same. However the decision process of how to analyze a connection is rather hard-configured into the NIDS. This means, at run-time the NIDS cannot adapt the analysis per connection. We develop a framework for connection oriented NIDS to decide at run-time per connection what analysis to perform. We use this framework for dynamically performing the appropriate application layer protocol decoding. Using this enhancement, a NIDS is for example able to reliably detect applications not using their standard ports, do payload inspection of FTP data connections and to reliably detect IRC based botnet clients and servers.

If a NIDS alerts in high volume environments, it does not have the possibility to provide a lot of context to the operator. For trading off disk space against forensic capability, we develop a NIDS-supplementary tool called "Time Machine". The tool records a prioritized yet comprehensive packet trace in high level environments. Our approach leverages the heavy tailed connection size distribution: it prioritizes small connections over large ones which greatly reduces the volume of traffic to record while retaining the largest fraction of the connections transferred.

For our operational evaluation of the tradeoffs and the developed mechanisms, throughout this thesis, we use the open source NIDS Bro. Its design is targeted to maximum flexibility, which makes it an ideal platform for powerful extensions and for use in a wide range of experiments.

Acknowledgments

Many people have contributed directly and indirectly to this thesis. Without their support, cooperation and guidance this work would not have been possible.

First and foremost I want to thank my advisor Anja Feldmann. Becoming a member of her research group at the Technische Universität München made it possible for me to enter the world of scientific research. She has always provided me with motivation, guidance, valuable feedback and support. In all research projects I have greatly benefitted from her broad knowledge in the networking area.

Also, my thanks go to my co-advisor Vern Paxson. During my visits at the ICSI and beyond, he inspired me with his visions and ideas. He always found time to discuss problems and helped me a lot to define the direction of my research efforts.

Furthermore I would like to thank Thomas Fuhrmann and Helmut Veith for agreeing to be on my thesis committee. I am particularly grateful for their great flexibility and organizing efforts.

Thanks also go to my colleagues and friends in our research group at TU München. They all contributed to a very enjoyable working atmosphere and it was fun to work together in various projects where all of us learned a lot. Special thanks go to Robin Sommer and Stefan Kornexl for their hard work in the projects we pursued together. With his ideas and his feedback Robin also greatly helped to improve this thesis. Also many thanks go to Petra Lorenz and Jörg Wallerich: Petra for always supporting me if anything had to be organized and Jörg for his mere endless endurance with managing our FreeBSD sniffer-park. Thanks to Vinay Aggarwal for all the illuminating discussions on Indian culture which along the way helped me to improve my English. Last but not the least I would like to thank the students that worked on our projects. Special thanks go to Michael Mai, who contributed a lot of programming work to our framework for dynamic analysis in NIDS.

Much of this work would not have been possible without the generous support of the people at the Leibnitz Rechenzentrum (LRZ) in Munich. They were not only willing to give us access to the network data of the MWN but also gave me insight into the security concerns in large network environments. In particular I want to thank Victor Apostolescu and Detlef Fliegl for their efforts to provide us with access to live network data and to let us participate in their testing of IPS appliances.

I also want to thank the Bro developer team, for their contributions to Bro and the discussions that greatly influenced this work. Also I am grateful to all the people at the ICSI that did a great job in supporting me before and during my visits. The close cooperation with the ICSI was enabled by a grant from the Bavaria California Technology Center, for which I am grateful.

Finally, I want to thank my family and friends. Without the love, support and encouragement of my parents, none of my achievements would have been possible. I am also indebted to Ute for the inspiration our relationship gave to me. I would like to thank her for her patience in letting me share my time with this piece of work. The support and views of my close friends, namely, Clemens Ruttloff, Diane Drescher, Christian Petersen, Andreas Merkel, Claus Huth and Thomas Krämer gave me a lot encouragement and our undertakings were always welcome for a change.

Contents

List of Figures

1 Introduction

1.1 Motivation

Today network intrusion detection systems (NIDS) are an indispensable component of most security frameworks. Their role is to monitor the network environment for security incidents. These systems examine network traffic for abuse based on security policies. Security policies define what to be considered abuse and have to be implemented in the NIDS. Once such a system identifies a policy violation, they either notify the network administrator so that he can take countermeasures or they block the malicious traffic themselves by active intervention. NIDS should be real-time systems as the reaction to a detected security policy should occur promptly to increase the effectiveness of the countermeasures.

In order to flexibly detect violations of almost arbitrary security policies, NIDS have in the last 10 years evolved into complex systems. They offer many different analysis forms, ranging from pure byte-string matching per packet to complete reassembly of communication streams and checking it against a precise protocol specification. Furthermore each analysis form can be tweaked with many parameters that influence the detection capabilities as well as the performance of the NIDS both in terms of CPU usage as well as memory usage.

Further complications in NIDS deployment are added by today's large-scale network environments. NIDS face extreme challenges due to traffic volume and traffic type diversity. High-bandwidth network connections do not allow the NIDS to take a long time for analyzing the packet stream in real-time. For example a 1 gigabit per second link can easily accommodate 100.000 IP packets per second. In this case, the system is allowed to use 10 microseconds in average for all packet handling and analysis if it wants to keep up with the traffic stream.

Deploying any of the current NIDS operationally in a large network environment, one quickly finds, that almost all of them can easily exceed the available resources in order to perform the desired analysis. Therefore we conclude, that a NIDS operator, when adapting the NIDS to his network environment, inevitably faces tradeoffs between analysis depth and resource usage: Given a set of resources, he has to decide if the system should look at more of the traffic at a coarse grain level or analyze some of the traffic at a deeper level of detail. This problem is aggravated by the fact that different analysis forms require different amounts of resources. Additionally most analyses offer many parameters with which to tune their resource consumption.

The tradeoffs between resource usage and analysis depth span a range from *predictive* to *adaptive* to *retrospective*. Regarding the first, we examine and model the influence of network traffic characteristics on resource consumption. The goal is to predict what

resources will be needed in the future. For the second, we aim at dynamically deciding per connection what analysis to perform. This allows to adapt the analysis to best use our resources on the traffic we face. The retrospective aspect we examine by developing a technique to record as much of the traffic as possible to allow more intensive analysis after a security incident.

Currently NIDS administrators tackle the problem of trading off detection quality against performance of a NIDS using a trial-and-error process. This process is time consuming and difficult due to the large number of parameters and the changing traffic characteristics. To overcome this limitation we provide a methodology for predicting resource usage. This can be used both, to predict when the current NIDS hardware resources are no longer sufficient given the current trend in traffic growth, and to determine a sensible starting configuration for unknown network environments.

If a NIDS performs unsuitable analysis for a part of the traffic, it wastes valuable resources. Therefore, a NIDS needs to be able to decide dynamically which analysis is suitable for the current traffic. This allows both, better detection quality and more dynamic resource management. More concretely, a close examination of common security policies shows, that almost all of them refer to applications. Many NIDS analysis forms depend on some knowledge about the application responsible for the network traffic.

This implies that the first task of an NIDS should be to determine which application is responsible for a specific subset of the traffic and then, if applicable and sufficient resource are available, perform the appropriate analysis. Currently NIDS tackle this problem by presuming that the network port information can be used to determine the application, e.g., all traffic on port 80 is Web traffic. This is no longer correct: Our experience in large network environment shows, that there are quite a few applications that do not use standard ports. As not all of these occurrences have malicious background, it is not only important to detect application layer protocols on non standard ports but also allow analyzing their payload appropriately. Therefore we developed a framework for NIDS that can use arbitrary protocol detection techniques to enable the NIDS to dynamically assign various analysis techniques to traffic subsets.

Especially in large-scale networks, the analysis depth vs. resource usage tradeoff of a NIDS has to be chosen to do rather coarse, quick analysis in order to keep up with the network speed. When a NIDS running such a configuration alerts, the operator has very limited information on the context of the alert. The most comprehensive context information would be to have a full packet-level traffic record at hand. Having the network traffic recorded would allow to perform deeper and more resource intensive analysis on just the suspicious traffic subset. This analysis could be done either completely offline and manually or automatically triggered by the NIDS as a reaction to suspicious rather than malicious behavior. Unfortunately bulk recording all network traffic in an environment where deep analysis is not possible due to resource constraints is also infeasible due to the sheer volume of data. To overcome this we in this thesis discuss how to leverage network traffic characteristics to drastically reduce the volume of traffic to be recorded.

An obvious challenge is to keep that part of the traffic that is important for security analysis. Regarding this, we rely on our experience, that the first few kilobytes of a connection most likely allow insight on the intent of the connection.

Throughout this thesis we use the NIDS Bro as a vehicle for implementing, evaluating and extending our analysis techniques. For this we leverage the modular and sound design of this open source system. It allows us to flexibly instrument and modify existing components as well as adding new ones. Many of our extensions are now part of the main Bro distribution.

1.2 Outline

The main contribution of this thesis is presented in three parts: In the first part, we examine the dependency between NIDS resource consumption and analysis depth. In the second part, we enhance NIDS analysis capabilities by overcoming the port based protocol analysis. In the third part we discuss how network traffic characteristics can be leveraged to comprehensively record traffic for security forensics.

Here we briefly summarize the contents of the following chapters.

In Chapter 2 we shortly discuss relevant background information on network intrusion detection in general and the specific NIDS Bro. As we use Bro as platform for our experiments and prototyping our ideas, we introduce Bro's design goals and analysis approaches in detail.

In Chapter 3 we examine the particular needs of today's NIDS in large-scale network environments. We offer an evaluation based on extensive operational experience. More specifically, we identify and explore key factors with respect to resource management and efficient packet processing and highlight their impact using a set of real-world traces. On the one hand, these insights help us to identify and gage the tradeoffs 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 the state management considerably as well as balance the processing load more dynamically. This enables us to operate a NIDS successfully in our high-volume network environments.

In Chapter 4 we shed light on the dependency between network traffic characteristics and NIDS resource consumption. For this we develop and validate a performance model for the NIDS Bro. We use this performance model to automatically determine a working parameter set for the NIDS in an unknown environment by analyzing rather low volume, connection sampled packet traces. Furthermore we develop a methodology to predict Bro's performance based on network traffic information abstracted to the connection level.

In Chapter 5 we develop a new technique for making the analysis of a NIDS dynamic for increasing the NIDS' detection quality: We discuss the design and implementation of a NIDS extension to perform dynamic application-layer protocol analysis. For each connection, the system first identifies potential protocols in use and then activates appropriate analyzers to verify the decision and extract higher-level semantics. We demonstrate the power of our enhancement with three examples: reliable detection of applications

not using their standard ports, payload inspection of FTP data transfers, and detection of IRC-based botnet clients and servers. Prototypes of our system currently run at the border of three large-scale operational networks. Due to its success, the bot-detection is already integrated into a dynamic inline blocking of production traffic at one of the sites.

Chapter 6 introduces a system that may augment NIDS for non real-time analysis or security forensics. We describe the design and implementation of a *Time Machine* to efficiently support recording and retrieval of full packet traces of high-volume network environments. The efficiency of our approach comes from leveraging the heavy-tailed nature of network traffic: because the bulk of the traffic in high-volume streams comes from just a few connections, by constructing a filter that records only the first N bytes of each connection we can greatly winnow down the recorded volume while still retaining both small connections in full, and the beginnings of large connections which often suffices. The system is designed for operation in Gbps environments, running on commodity hardware. It can hold a few minutes of a high volume stream in RAM, and many hours to days on disk; the user can flexibly configure its operation to suit the site's nature.

1.3 Published Work

Parts of this thesis have been published:

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

Stefan Kornexl, Vern Paxson, Holger Dreger, Anja Feldmann, Robin Sommer
Building a Time Machine for Efficient Recording and Retrieval
of High-Volume Network Traffic
Proc. of the 5th ACM SIGCOMM Internet Measurement Conference, 2005

Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, Robin Sommer
Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection
Proc. of the 15th Usenix Security Symposium, 2006

2 Background

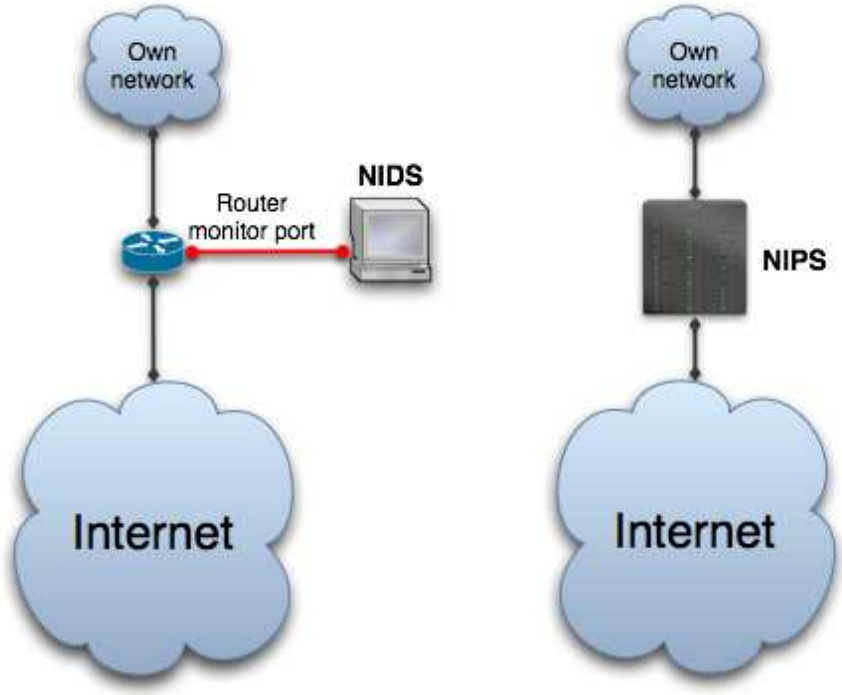
This chapter presents background information used in the rest of this thesis. First, we give a short overview on network intrusion detection in general, its goals, its techniques and its problems, then we present a concrete NIDS in more detail: Bro. An overview on the network environments, we had access to for developing, testing and evaluating our intrusion detection techniques concludes the chapter.

2.1 Network Intrusion Detection

System operators have to protect computer systems in their care from abuse. To achieve protection, they have to harden the systems and limit access to the resources. How much and what access has to be restricted is defined in a *security policy*. The security policy of a site can be seen as a set of rules users have to obey.

On the network as access medium, system operators can resort to a multitude of active *security devices* like packet filters (i.e. firewalls) or other access control mechanisms. On the other hand, as no protection plan is perfect, the network administrator wants to monitor that the security policy of the site is obeyed by the users and enforced by the existing security devices. That is where *Intrusion Detection Systems (IDS)* come into play. Their goal is to work as a "burglar alarm" for a resource: If an attacker is violating a rule (meaning he attacks or misuses a resource) in the environment, the IDS is supposed to alert the system operator. As with access control mechanisms, a fundamental problem is how to implement the security policy in an IDS: Security policies have to be translated into technical rules, the IDS is able to understand and check. Intrusion Detection Systems counter this challenge with complexity: The idea is that offering a large set of features enables the operator to effectively implement his policy.

IDS can be separated into two classes: *Host based Intrusion Detection Systems (HIDS)* and *Network Intrusion Detection Systems (NIDS)*. Host based Intrusion Detection Systems monitor the processes running on a single host for policy violations. In contrast, Network Intrusion Detection Systems monitor network packets going to and from all hosts in the network. Both approaches have advantages and disadvantages. HIDS enable a system operator to detect abuse done by users working locally on the host as well as users logged in remotely over the network. In contrast, NIDS can only detect attacks carried out over the network. On the other hand one NIDS can monitor many hosts at once and correlate attacks targeted at several hosts without implying any instrumentation on the hosts itself. In today's IDS landscape there are also hybrid systems: Typically NIDS that augment their analysis with input from host sensors. In this thesis our focus is on network intrusion detection systems and the techniques they deploy.



(a) (b)

Figure 2.1: Standard deployment of network intrusion detection systems

2.1.1 NIDS Deployment

NIDS work by analyzing networks packets that are sent to and from the protected resources. In the most common case a NIDS is used to detect attacks on a local network with a single up-link to the Internet as shown in Figure 2.1. In this case the NIDS either has to be supplied with a copy of all packets traversing that link (see Figure 2.1 (a)) or it has to actively forward the packets (see Figure 2.1 (b)). If it is supplied with copies of the packets, it is considered to be ‘‘passive" or *"monitoring only"* whereas if it forwards the packets itself it is termed *"active"* or *"in-line"*. Today, these active systems are also reffered to as *Network Intrusion Prevention System (NIPS)*. If an in-line NIDS is detecting malicious behavior, it is able to block (that means not to forward) the malicious traffic. Deploying active NIDS as access control devices is controversial debated in the security community (e.g., [Bej04]). Such a system is “fail-close” as it in case of a crash or failure does not forward packets any more. This at the first glance may be a great advantage but it also means that an active NIDS does introduce a new single-point-of-failure: Deployment of such a system does only make sense if the system adds significantly to the protection of the resources but on the other hand does not hinder people to use the resources as they are supposed to. In other words: if the system often

blocks legitimate traffic or does crash regularly it is not suitable for productive network environments. Passive NIDS on the other hand are fail-open. If it triggers an alarm there is by default no automatic reaction. The operator may of course, after interpretation of the alarm, react but for many attacks human reaction times are not short enough to prevent an ongoing attack. Nevertheless, even without automatic reaction, the system may deliver valuable information to the network operator: knowing that a system has been compromised enables the system administrators to avert further damage by e.g., information theft.

In technical terms, the actively forwarding systems are easier to implement: The NIDS simply provides two network interfaces and is introduced as an additional router into the network link. However, as discussed above, the challenge is to provide enough throughput and keep the pass-through times low. For the passive approach there are two main options: The most popular technique is to configure a so called *monitor-* or *mirror port* on a router which duplicates all packets on the Internet access link. The alternative are so called *wire-taps*: physical devices which resemble something like a T-shaped pipe for network links. For more information on network tapping techniques and sample topologies refer to [Bej04].

2.1.2 Network Intrusion Detection Quality

The goal of a NIDS is to analyze network data in order to detect behavior that could compromise the network security, short attacks, in a given environment. Detection quality basically is defined along two metrics: *false negatives* and *false positives*: The former are missed attacks, meaning there was an actual attack but the IDS did not detect it e.g., due to inappropriate analysis of the data. The latter are false alarms. In this case the IDS issues an alert, but there was no security relevant threat. False positives are often caused by imprecise detection algorithms of the NIDS. On the other hand, unintentionally inappropriate but benign usage of network resources can also trigger false positives.

The problem that false negatives cause is apparent. The NIDS is "blind" to certain attacks, drastically reducing the value of a system that should protect resources against misuse. False positives on the other hand imply a different problem: Each alarm triggers some reaction: In the worst case an operator has to check whether the alarm makes sense or not. Like someone who is shouting warnings all the time about a fire that does not exist, the alarms of the NIDS are ignored by the IDS operator at some point. In the case the IDS discovers a real attack, it is likely that the alarm is ignored too. In the case that there is some kind of automatic reaction, e.g., blocking of the corresponding traffic, triggered by an alarm, false positives are fatal too: Alarms now turn to annoyed and complaining end-users which again may render the NIDS to be unusable for the network operators.

NIDS operators and developers obviously aim at reducing false negatives and false positives to a minimum. An ideal NIDS would detect every attack (no false negatives) and would never notify the administrator unnecessarily (no false positives). Unfortunately, reducing false negatives and false positives is extremely hard in reality. Bejtlich

reasons in [Bej04] that intrusion detection will never be 100% accurate since they lack context. He defines context to be "the ability to understand the nature of an event with respect to all other aspects of an organizations environment". More technically, Ptacek et. al. show in [PTN98], how NIDS analysis techniques can be deceived by manipulating network traffic so that the NIDS interprets it different than the actual end system. Modern NIDS come with techniques to counter these attacks e.g, traffic normalization [HKP01] or actively collecting information on the hosts to be protected [SP03a]. In [Axe99] Axelsson points out a more fundamental problem of IDS: They analyze huge amounts of data of which only a quite small fraction is actually malicious. In the paper Bayesian statistics are applied on a typical ratio of malicious and benign "input-events". The conclusion is, that even a system that has no false negatives at all needs to have a very low false alarm rate (i.e. 1×10^{-5}) in order to achieve substantial values of the Bayesian detection rate (that is the probability of an intrusion under the condition that there is an alarm).

2.1.3 Network Intrusion Detection Techniques

Network Intrusion Detection Systems can be classified into three categories each using a different approach of detecting attacks. The two traditional approaches are called *misuse detection* and *anomaly detection*. A rather new approach is the so called *specification-based detection*. We now take a closer look at each technique and it advantages and disadvantages.

Misuse detection is based on a definition of misuse. That means the behavior that is considered to be dangerous or compromising has to be described to the NIDS. The NIDS then compares the current usage of the resources with the misuse usage patterns and alerts on matches. Most NIDS incorporate misuse detection by implementing *signature matching*. A signature, in this context, is a characteristic byte pattern of a known attack. What attacks can be detected by signatures and how good the detection quality is, depends among others on how exact the characteristic attack patterns can be described. Most NIDS nowadays allow signatures to use regular expressions for describing the byte patterns. This technique adds significantly to detection quality [SP03b]. On the other hand, powerful matching capabilities are no guarantee for high signature quality. To our experience, for the open source NIDS snort [Roe99] many signatures have poor quality, resulting in a lot of apparent false positives. Nevertheless, given signatures of good quality, meaning signatures that tightly describe a characteristic misuse pattern, the resulting low rate of false positives is the huge advantage of misuse detection. The most significant disadvantage of misuse detection is the conceptual inability to detect unknown attacks.

The second traditional approach to intrusion detection is anomaly detection. As is apparent from the name this technique works by distinguishing normal behavior from non-normal behavior. For anomaly based NIDS the idea is, that traffic containing an attack looks different than normal traffic. Having a knowledge base of normal behavior patterns, the IDS compares certain characteristics of the current behavior with the corresponding characteristics of the normal behavior from the knowledge base. If the

deviation between the normal and the current behavior exceeds some threshold, the IDS issues an alarm. In practice there is a wealth of heuristics that implement anomaly detection for NIDS. Usually statistic methods are used to gain an abstract view on the network traffic that in turn can be compared to the same statistic view on normal network traffic. A popular example for a statistic metric is "transferred volume per time interval". For every time interval of size t the transmitted traffic volume v is measured and compared against what is considered the normal transmission volume v_{norm}. The fundamental problem of anomaly detection can also be seen in this example: How to get an appropriate value for v_{norm}; or more generally: How to define normal behavior? For the example outlined before one could compare v against the volume v' measured at the same time of day exactly one week before. In this case one would assume that the volume v' resembled normal behavior. The advantage of anomaly detection is clearly the ability to detect unknown attacks. On the other hand the extreme variability of regular network traffic makes it very hard to come up with statistical metrics that are stable as long as everything behaves normal but show significant deviation as attacks take place. This usually results in a high false positive rate since regular traffic variability often causes significant deviation in typical statistical traffic metrics.

The third approach, specification based detection, aims at combining the advantages of misuse detection and anomaly detection. The idea is that the operator specifies the allowed behavior manually. Everything that occurs and is not specified violates the security policy and is therefore considered an attack. The advantage of the approach is that detection quality is high. By doing a specification derived from the security policy false negatives as well as false positives can be minimized. The disadvantage is, that it is a very labor-intensive process to *(i)* map out a comprehensive security policy and *(ii)* generate a tight specification for that policy. Furthermore, generating the specification once is not enough. Especially in large network environments the policy and the specification have to be continuously adapted to the network usage profile.

2.2 Bro

The NIDS Bro was developed by Vern Paxson. It began continuous operation in 1996, with the first paper describing it published in 1998 [Pax99]. Bro is designed to be a real-time network intrusion detection system. Its design also aims at making it very flexible and extensible. In particular, it is conceptually not restricted to any of the above detection techniques alone. Its basic mode of operation as presented in [Pax99] is deep protocol analysis. For this, an important design criterion in Bro is that the system should anticipate attacks against itself. Its analysis components are built so that they detect and flag evasion techniques as good as possible. In its early stage, it was able to analyze IP, TCP, UDP and six application layer protocols. The modular design allows specialized application layer protocol analyzers to be added easily. So today, Bro has analyzers for 16 widely-used application layer protocols, among them HTTP, FTP and IRC.

The abstract model of Bro is shown in Figure 2.2. The main idea behind the layered

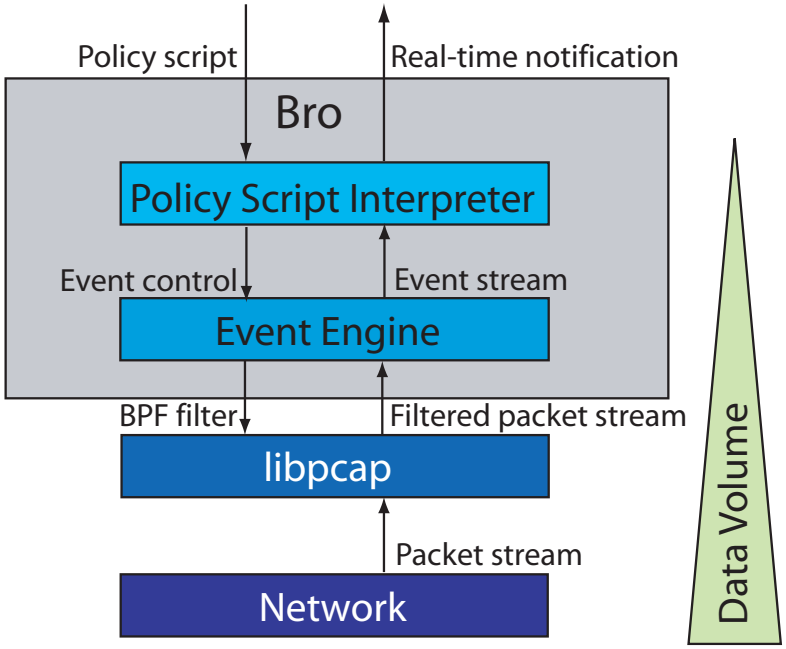


Figure 2.2: Structural design of the Bro NIDS

model was to separate mechanism from policy. This results in different levels of traffic abstraction each layer. The lower layers get the bulk of the traffic to analyze. That means they have to be optimized for speed and packet throughput. The upper layers get higher-level aggregated and/or filtered input and can therefore afford to spend some more time on closer analysis or correlation of the input entities. The operator interacts only with the upper layer by implementing his policy. We will now briefly describe more details on each layer.

2.2.1 The Filter Layer

The lowest two layers actually deal with network packets. The filter layer is not implemented in Bro itself. It rather uses the facilities of the libpcap library [Lib]. Using libpcap, Bro abstracts away from the details of the underlying link layer technology. Even more important, libpcap deploys the kernel filter provided by the operating system to filter out packets, if available. What packets it discards is configured in a user supplied BPF filter expression [MJ93]. For Bro, the filter (the decision, which packets to look at) is defined by the upper layers so that those packets that are not analyzed anyway are dropped. Dropping not-analyzed packets that early helps to reduce CPU time spent on "uninteresting" packets. Figure 2.3 shows an example for a filter deployed by Bro. In this example Bro is configured to analyze ftp and telnet on the application layer plus to perform basic TCP connection analysis. The filter discards any packets that are not sent from or to port 21/tcp or 23/tcp. As 21/tcp is the IANA assigned port for ftp control

```
port telnet or port ftp or tcp[13] & 7 != 0
```

Figure 2.3: Example BPF filter expression generated by Bro

connections and 23/tcp the one for telnet, Bro is able to perform the desired analysis without wasting resources on the rest of the traffic. The last part of the filter expression allows Bro to perform basic TCP connection analysis: It enables those TCP packets to pass the filter that do have at least one of the TCP flags `SYN`, `FIN` or `RST` set. These special packets control TCP connection establishment and teardown. In the rest of this document we term them *TCP control packets*.

2.2.2 The Event Engine

Bro itself consists of the layers "Event Engine" and "Policy Layer". The event engine is written in C++ and performs a policy neutral analysis of the network packets that pass the filter layer. Policy neutral means the event engine does not make any decisions as to what constitutes an attack. It just generates a more abstract view of the network traffic by parsing, analyzing and correlating the single network packets. The result are so called *events* which contain the semantic structure of the traffic.

The basis for any further analysis in Bro is that each packet is associated to a connection. The connection-less protocols UDP and ICMP are also coerced into a connection semantic. In order to interpret all packets as part of connections, the event engine has to parse the IP, ICMP, TCP and UDP protocols. The event engine's task is to apply basic per packet sanity checks like checksum verification but also performs stateful analysis like IP fragment reassembly and connection-handling. For TCP, Bro implements a state machine for each connection: For each TCP packet, the corresponding TCP connection object is looked up or created. Then its state, e.g., last seen sequence number or the connection state in the state machine is updated. Events are for example generated when a connection changes its state to established (event `connection_established`) or terminated (event `connection_finished`). If a connection request is not answered for some time Bro issues a `connection_attempt` event. This is realized by timer objects associated to each TCP connection. If the data will be further analyzed by some application layer analyzer (see below), the single TCP packets belonging to a connection will also be assembled to a payload stream.

For application layer protocols, Bro features a set of protocol specific *analyzers*. Each analyzer has functions for parsing "its" application layer protocol and generates additional events just for connections using that protocol. Analyzers that were included from the beginning in Bro are for example the *ftp* and the *telnet* analyzer. Both take the reassembled TCP payload as input and generate application specific events. The FTP analyzer e.g., issues a `ftp_request` event each time a line containing a FTP request from the FTP client is completely sent to the server. The telnet analyzer generates e.g., a `login_success` or `login_failure` event upon successful/unsuccessful user authentication. In general, a specific event is only generated, if they are used for the

implementation of the site's policy.

Based on its connection handling, Bro was extended by a signature engine [SP03b] in 2003. Bro's signatures are defined as regular expressions. Additional to the enhanced power of regular expressions over simple byte-string signatures, Bro's signatures can be augmented with Bro's rich internal state about connections and application layer protocol analysis. In particular, Bro's signature matching engine leverages Bro's connection stream reassembly and allows to define signatures for special application layer protocol header fields (e.g., the HTTP URL) and bidirectional signatures. Bidirectional signatures define two linked "subsignatures": one for each direction of a connection. A bidirectional signature matches only if both subsignatures match. Upon signature match policy neutral events are generated as for any other analysis on the event engine level.

2.2.3 The Policy Script Layer

The policy script layer adds the site-specific interpretation of the so far policy neutral events: The operator specifies the site's policy using a specialized, richly-typed high level language. He leverages the traffic analysis done in the event engine by implementing reactions on the events generated there. These *event handlers* are only the entry points into the policy layer. In general, the operator is allowed to maintain an arbitrary amount of state which he has to manage explicitly. For example a simple policy about scans could generate a notification if one source unsuccessfully attempted to contact a certain number of hosts. It would implement a event handler for the `connection_attempt` event and maintain a table of contacted destinations per source. Source and destination IPs of the connection attempts are supplied by the event engine via parameters. Policy scripts can send out real-time alerts, record activity transcripts to files and execute programs as a means of reactive response.

In general one can say Bro is configured by the policy scripts. These scripts "decide" what analysis the event engine performs by implementing the respective event handlers. Additionally the event engine "exports" variables that influence the protocol analysis in the event engine. The values of these variables are defined in the policy scripts. An example is the time after which a connection attempt is considered to be unsuccessful. Also, the BPF filter is specified by the policy scripts. That is, the user defines what part of the packets the system gets to analyze.

The separation between mechanism and policy (event engine and policy scripts) in Bro can be seen as a variant of *Kerkhoff's principle*. A smart attacker is supposed to analyze Bro's mechanism part, that is the analysis code in the event engine. However the site's policy, that is its parametrization of the analysis, has to be kept secret by the operator. Thus, the attacker knows what analysis is possible to do, but evasion is more difficult, since he can only guess about how the analysis is done exactly.

2.2.4 Extensions

Extending Bro usually works by adding policy neutral parts of a new analysis to the event engine and implementing environment specific interpretation and parametrization

in policy scripts. Extensions are for example new protocol analyzers. A parser part generates new events and the policy script defines corresponding event handlers. Programming protocol parsers manually is a tedious and error-prone process. The witty worm [SM04, KPW05] is an example for the consequences of a buffer overflow vulnerability in the analyzing code of several *Internet Security Systems (ISS)* [ISS] products. Pang et. al. present in [PPSP06] a declarative language and compiler for constructing semantic analyzers for complex network protocols. The language allows a developer to concentrate on the high-level aspects of a network protocol, while at the same time achieve correctness, robustness, efficiency, and reusability of the code.

There are also extensions to Bro that use anomaly detection techniques. In [ZP00a] and [ZP00b] Zhang and Paxson present techniques for detecting interactive backdoors and stepping stones. They implement their novel algorithms into Bro. The algorithms base on the traffic characteristics directionality, packet sizes and packet interarrival times. As both approaches do not rely on packet contents, they are able to detect backdoors respectively stepping stones by analyzing encrypted connections.

Another algorithm presented in [JPBB04] is also implemented in Bro. It uses anomaly based detection techniques for fast portscan detection. The authors use sequential hypothesis testing to determine whether a user accesses hosts corresponding to the "good user" model or rather the "bad user" or scanner model.

Sommer and Paxson present in [Som05] a major extension to Bro which enables the system to leverage "independent state" for various applications. Independent state is internal state of a NIDS that can be propagated from one instance of the NIDS to another. Among the discussed and evaluated applications are distributed processing, load parallelization, and selective preservation of state across restarts. For this Bro is extended by a communication protocol. This protocol is also implemented in `libbroccoli` [KS05] limiting its use no longer to "state exchange" between Bro instances but allowing "state injection" as for example done in [DKPS05]. This work discusses a technique to support Bro with host based context. This helps for example to resolve ambiguities in the interpretation of the end system as is demonstrated in the paper for HTTP requests.

2.3 Environments

The basis for our studies is our operational experiences monitoring a heavily-loaded network environment, the Münchener Wissenschaftsnetz (*MWN*), Germany. During the main time of our studies, until begin 2006, the MWN had a Gigabit per second up-link to the "Deutsches Forschungsnetz", (DFN), the national Internet provider for all scientific institutes in Germany. Then the link was upgraded to a 10 Gigabit per second up-link.

The MWN provides Internet connectivity to 2 major universities, the Technische Universität München, the Ludwig-Maximilians-Universität München and quite a large number of research institutes. Overall, the network contains about 50,000 individual hosts and 65,000 registered users. On a typical day, 1–3 TB of data is transferred, which averages to 44,000 packets/sec. The average utilization during busy-hours is about 350 Mbps

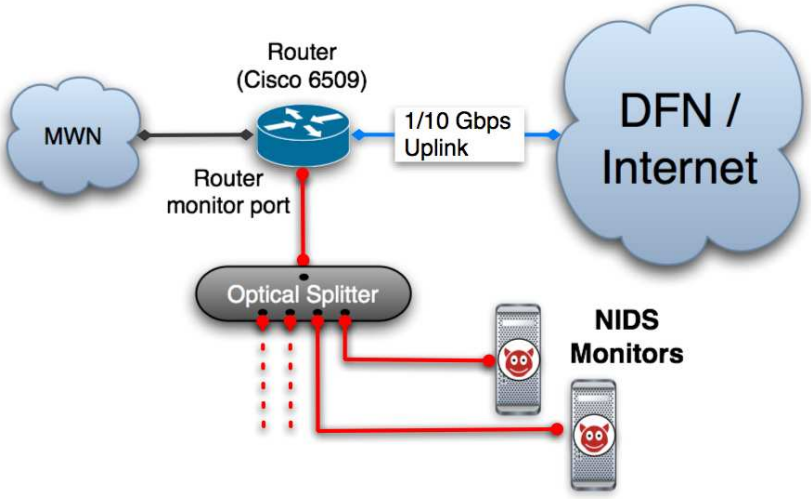


Figure 2.4: NIDS monitor setup in the MWN environment

(68 Kpps). Usually, most of the connections are HTTP (65–70%), while the biggest contributor to traffic volume is FTP (about half of the total volume). Alone 15–20% of the transmitted traffic volume comes from the popular FTP mirror `ftp.leo.org` hosted by the Technische Universität München. The network is configured to *block* well-known peer-to-peer ports, along with certain other services—primarily SNMP, NetBIOS/SMB, and Microsoft SQL. The blocking of the ports of popular peer-to-peer systems explains that HTTP dominates the traffic mix.

Our setup for monitoring all traffic being exchanged between the MWN and the Internet is depicted in Figure 2.4. Our monitor-farm is connected via a Gigabit Ethernet link to a port of the MWN's upstream router, a Cisco 6509. This port is configured to passively mirror all packets either leaving or arriving on the outbound interface. Note, that this setup may induce packet drops on the router: The full-duplex 1 Gbps access link has double the bandwidth (1Gbps *towards* the DFN and 1Gbps *from* the DFN) of our monitor link. In practice, we do indeed see rare drops caused by short time traffic peaks.

At our site we duplicate the packets using a passive optical splitter. The fibres with the monitored packets are connected to the actual monitoring PC systems. For our studies we used Dual Athlon MP 1800+ with 2 GB Memory, running FreeBSD as primary NIDS monitors. Concurrent studies in our environment [Sch05] showed, that these systems proved to have the best performance for packet capture and processing.

For some of our studies we also refer to experiences gathered at three additional high-volume network environments. While all these institutions also transfer large volumes of data (one to several Terabytes a day), their networks and traffic composition naturally have qualitative differences. All three additional networks are academic network envi-

ronments and located in California, USA. The institutions that deploy these networks are the Lawrence Berkeley National Laboratory (*LBNL*), the University of California, Berkeley (*UCB*), USA and the National Energy Research Scientific Computing Center (*NERSC*).

- **UCB:** The transferred traffic volume at UCB is with 3–5 TB per day usually even higher than in MWN. UCB uses three 2 Gbps up-links to the Internet to serve about 45,000 hosts on the main campus plus several affiliated institutes. In contrast to the MWN, UCB has a no-filter policy, resulting in a much higher usage of peer-to-peer applications than in MWN.
- **LBNL:** The Internet up-link in the LBNL environment is, like in the MWN, a 1 Gbps link. The transferred volume is with ca. 1.5 TB per day quite a bit lower as is the user population of 13,000 users. However, the busy-hour load in this network is, with approximately 320 Mbps (37 Kpps) similar to that of MWN.
- **NERSC:** The National Energy Research Scientific Computing Center (*NERSC*) is administratively part of LBNL, but uses a physically separate access link. As one of the largest supercomputing centers of the USA it provides computational resources (around 600 hosts) to 2,000 users. The network traffic at that site is dominated by large transfers, containing significantly fewer user-oriented applications such as the Web. The busy-hour utilization of the Gbps link is 260 Mbps (43 Kpps).

In all three environments, the monitoring machines are connected to the Internet access link as in MWN. The technical setups for packet monitoring use router mirror ports (UCB) and direct fiber taps (LBNL, NERSC).

3 Operational Experiences with High-Volume Network Intrusion Detection

3.1 Introduction

The practical experience of running a network intrusion detection system (NIDS) operationally is that with increasing traffic volume the challenges grow. Three major difficulties arise. First, the sheer packets-per-second (pps) rates can reach levels at which the load due to interrupts and filtering push the system into thrashing. Second, as volume rises – particularly if it rises due to greater numbers of hosts – so does the traffic's *diversity*, which can stress the NIDS's fidelity by generating both more false alarms and a wider range of types of false alarms. Third, as the number of hosts increases, so does the burden of managing *state* and other resources.

These practical difficulties with high-volume network intrusion detection 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 high-volume, operational environments.

In this chapter, we offer such an evaluation. Our study is in the context of using commodity PC hardware running open-source software for operational security monitoring of quite high-volume environments (Gbps, 10s of thousands of hosts transferring 2-3 TB/day). We found that in such environments, if we simply install and run an untuned/uncustomized NIDS such as the open-source Snort [Roe99] or Bro [Pax99] systems, they are unable to effectively cope with the amount of traffic. Snort immediately consumes the entire CPU, leading to excessive packets losses, while Bro, in addition, quickly exhausts all available memory.

Obviously, the volume is too great a burden for the NIDS. But what are the key factors that lead to such severe difficulties? In this study we look at a number of issues that arise due to the problems of resource management and efficient packet capture and filtering. We aim to analyze the main contributors to CPU load and memory consumption and look for means to ameliorate their impact, which sometimes requires developing new mechanisms if the available tuning parameters do not suffice.

For a *stateless* NIDS, the load imposed on the CPU is the main limiting factor. This load is correlated with the types of analysis as well as the traffic's volume and makeup. A *stateful* NIDS, additionally, maintains an in-memory representation of the current state of the network, which must be meticulously maintained at all times. This state provides

the context necessary to evaluate the network events. Like CPU load, the volume of the state is also correlated with the traffic volume as well as the types of analysis, and is constrained by the system's available memory. Since maintaining state requires state management, the NIDS requires some significant CPU time just for updating data structures.

Common approaches for limiting NIDS resource usage include different kinds of state management (e.g., via timeouts and/or fixed size buffers); checkpointing [Pax99] (i.e., regularly restarting the system to flush old state); limiting the traffic by analyzing only certain protocols or subsets of the address space; and distributing the work to multiple machines. To understand the efficacy of these approaches, we examine the resource requirements of a NIDS and the associated tradeoffs *in operational use*.

Our study is in the context of the Bro NIDS, which we have deployed operationally in a couple of high-performance environments. Recall, that Bro is a highly stateful NIDS: In its two-layer architecture, both the event engine layer and the policy script layer generate and manage a great deal of state.

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 (fluctuating) per-packet processing time. While these factors certainly are not surprising by themselves, the key is understanding the tradeoffs between them with respect to tuning a NIDS and adapting it to the environment. In addition, we found several new ways to reduce resource requirements, considerably improving state management and dynamically balancing the processing load. While the concrete realization of these is tied to the particular system we examine, the underlying concepts are applicable to other NIDS as well.

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. We find that to understand memory usage and CPU load spikes, particular care is needed to soundly instrument the system; the somewhat atypical tradeoff between CPU and memory vs. detection rate, and the sensitivity of such a system to quite small programming errors is rather illuminating; furthermore, a high-volume monitoring environment can exhibit artifacts that significantly affect any analysis.

In Section 3.2 we summarize related work. After describing the main high-volume environment that we use for our study (see Section 3.3), we discuss our operational experiences and demonstrate certain effects using a set of traces (see Section 3.6). In Section 3.7 we present several enhancements to Bro which together enable us to now operate Bro successfully in high-volume environments.

3.2 Related work

Reports in the literature of operational experiences with high-volume network intrusion detection are quite rare. More generally, a major question for evaluation studies is what sort of traffic to use. [HW02] proposes a methodology to craft traffic with different

characteristics. But in high-volume environments, such characteristics are often unpredictable. Traffic patterns vary widely between different environments [FP01, Pax94], and Internet traffic includes significant short-term fluctuations [FGW98]. Moreover, 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]. In addition, attackers can target the NIDS itself to try to evade [PTN98] or overload the system [CW03, Pax99]. Tools like Snot [Sno] or Stick [Sti] craft packets to match known attack signatures, thereby stressing the NIDS's logging system.

To avoid overload, some systems distribute the analysis across multiple machines (e.g., [KV03, SP05]). This certainly can help, but the individual machines still face the fundamental problem of limiting and managing their resource usage. Along these lines, [LCT+02] presents an approach for adapting the configuration of a NIDS to the current load. By quantifying benefits and costs of analysis tasks, they dynamically determine the best configuration under given resource constraints. While our concept of load-levels (see Section 3.7.3) is similar in spirit, we find it quite hard to crisply define such cost metrics for high-volume traffic analysis (see Section 3.6.3 and Section 3.6.4). Thus, we statically define a set of configurations appropriate for medium-term traffic changes (which may not suffice under overload attack situations).

Most evaluations consider detection rate as their major performance criteria, gauging tradeoffs between false positives and false negatives. But from our experience we argue one must not lose sight of the fundamental tradeoff between detection rate and resource usage. It is rare that studies explore this consideration, and in fact often the particular configurations of evaluated systems are unclear. For example, for signature matching [SP03b] shows that alerts often depend on the underlying implementation and its concrete parameterization. For a general discussion of these difficulties and pitfalls, see [Ran01].

Using commodity hardware, high-speed packet capture is quite challenging [AGJT03]. A key factor is the architecture of the operating system's packet filter [Der03]. As on our monitors, we use FreeBSD, we are provided with an in-kernel implementation of the *Berkeley Packet Filter* [MJ93], giving us quite efficient, stateless packet filtering.

3.3 Traces

While we gained our experiences and insights from deploying the NIDS 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 monitor to demonstrate the challenges that a NIDS is facing:

The trace `mwn-week-hdr` contains all TCP control packets (SYN, FIN, RST) for a six day period. The compressed trace totals 73 GB, contains 365M connections, and 1.2G 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 2PM and 4PM. (Basic statistics: 13 GB compressed, 471M packets, 11M 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, 19M packets, 404K 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`, a popular Web server. (2.8 GB compressed, 38M packets, 1M connections of which nearly all are HTTP). `mwn-irc-ddos` is a 2.5-day trace including full payload from `irc.leo.org`. 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, 76M 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 is the monitor itself, which we mainly used for any analysis involving the large `mwn-week-hdr` trace. The others are 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.

The performance evaluations presented in the following sections use the measurement methodologies for system memory usage and run-time measurements presented in Section 3.4 and 3.5.

3.4 Measuring Memory Usage

If we want to reduce the memory usage of a stateful NIDS, we need to understand 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.

3.4.1 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 simulates the underlying processor 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.

3.4.2 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 always provides us with an upper bound for the application's peak allocation. In FreeBSD, on the other hand, we can allow the C library to return unused memory to the system, thereby decreasing the total heap-size. At the same time, FreeBSD's C library does not provide an easy way to access the current allocation. On Linux with glibc, 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 $n + p$, of which $p \geq 0$ are 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.

We note that in practice it is very hard to instrument a complex system accurately. Therefore, values delivered by internal instrumentation are often only estimates 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 particular chunk of memory. 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 (on average, we are able to classify about 90% of the memory allocation; the rest is allocated at locations that we did not instrument).

In the main text, *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 lower bounds. These values *include* malloc()'s padding and assume a glibc based system. When we give the memory allocation for a particular trace, we always refer to the *maximum* for this trace.

3.5 Measuring CPU Usage

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

3.5.1 External Tools

An 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 more fine-grained insight, but their overhead is much too large to infer real-time behavior.

3.5.2 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 say 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 which 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. For example, Linux and FreeBSD's `getrusage` system calls provide a default resolution of approximately 10ms on Intel hardware. 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 timing 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 when 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 $n = 10,000$, giving us times t within 30–50ms with Bro's minimal configuration (on an Athlon XP 2600+). Figure 3.5 shows 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 get perfect values of real-time CPU usage. But our measurements give us some very valuable intuition on the system's behavior. Part of our ongoing work is to flesh out the model further (see Chapter 4).

3.6 Operational Experiences

Deploying NIDSs operationally in our high-volume environments presents several different challenges. Most problems announce 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., of analysis depth or of parameter values, faces a tradeoff between quality (i.e., detection rate) and quantity (i.e., required resources).

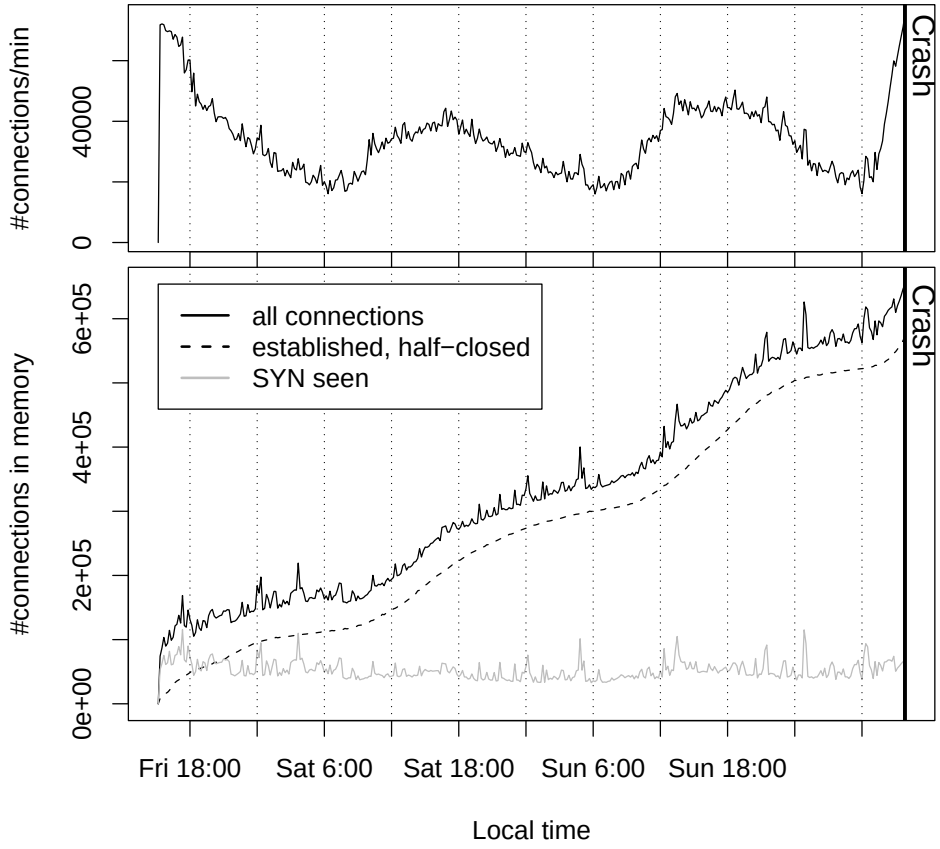


Figure 3.1: Connection state from `mwn-week-hdr`, default configuration

In this section we discuss major issues that had to be addressed: state management that is either too liberal, or not existent at all; data and processing peaks causing missed packets; and small programming deficiencies causing major problems. Next we recapitulate a recurring experience: in network intrusion detection, one faces a rather unusual tradeoff between resource requirements and detection rate. Finally we conclude the section outlining some problems due to the monitoring environments rather than the NIDS itself. We discuss various mechanisms that allow us to overcome these difficulties in Section 3.7.

3.6.1 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 million TCP connections per day.

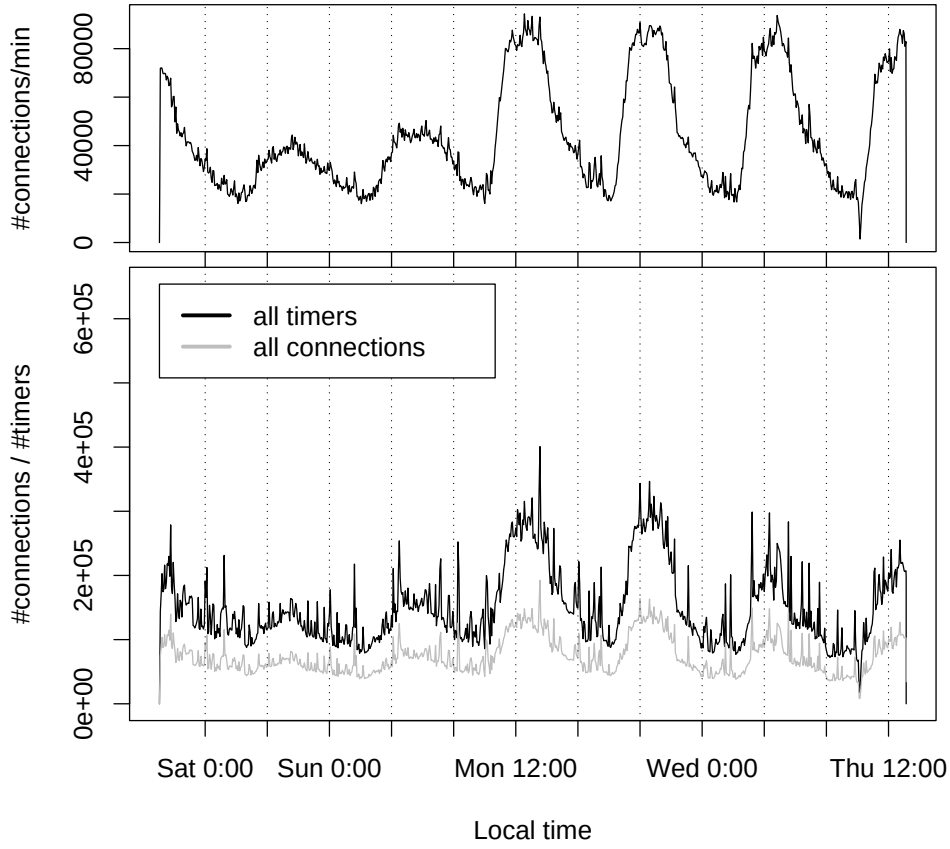


Figure 3.2: Connection state from `mwn-week-hdr`, inactivity timeouts enabled

The amount of memory required for connection state is determined by two factors: *(i)* the size of a 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 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-layer protocol analyzer for a connection, this can grow significantly. Run-

ning Bro in its default configuration[1] 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 3.1(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 3.1(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 in the "SYN seen" state does not increase dramatically, we conclude that the problem is not that Bro fails to 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 3.1(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. Instead, Bro's original, very coarse-grained state management approach is to periodically terminate the monitor, flushing all state, and then restart the analysis from scratch. Clearly, this approach degrades the quality of the analysis and provides easy evasion to attackers who split their attacks across restarts.

In accordance with the above design goals, 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 Sections 3.6.3 and 3.6.4), and *(iii)* artifacts caused by the monitoring environments (see Section 3.6.7).

Whatever the cause, however, accumulating connection state indefinitely over time is clearly not feasible. There are similar problems with UDP and ICMP "connections". Remember, that while UDP and ICMP are not connection-oriented, Bro uses a flow-like definition to fit them into its connection-oriented framework. Most of these are also never

[1] 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.

removed from memory. While there already is a way to mitigate these problems,[2] this still does not suffice. In Section 3.7.1 we develop an approach to mitigate this problem using inactivity timers. Figure 3.2 shows the success of including this extension.

3.6.2 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 3.3(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 using inactivity timeouts as described in Section 3.7.1.) Figure 3.3(top) again shows the number of connections seen per minute. We recognize from Figure 3.3(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. The "crud" [Pax99] 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 Section 3.7.2.

In Section 3.7.2 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 3.4.

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

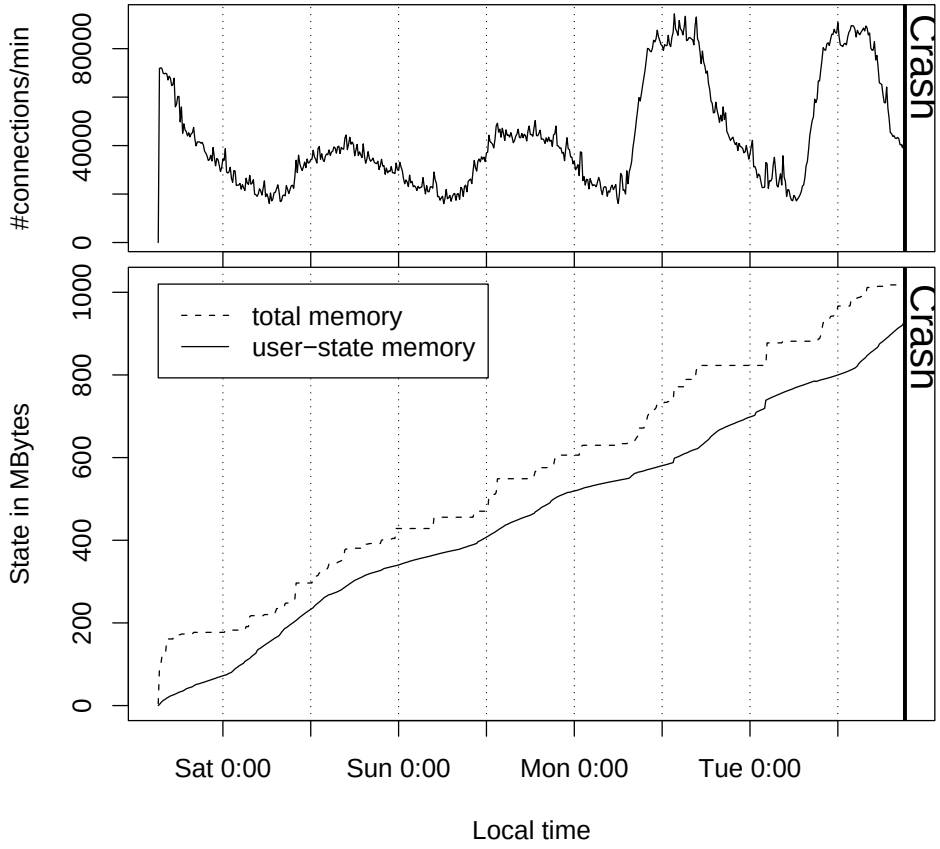


Figure 3.3: Memory required by scan detector on `mwn-week-hdr` using inactivity timeouts for connections. Default configuration.

3.6.3 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-

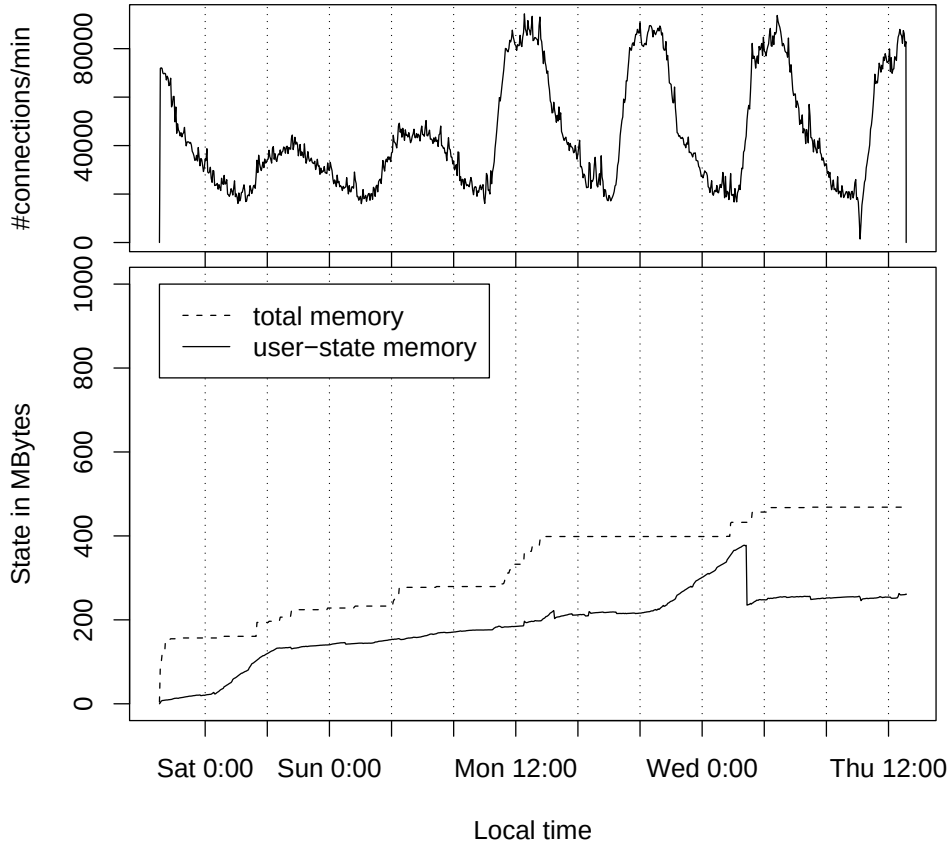


Figure 3.4: Memory required by scan detector on `mwn-week-hdr` using inactivity timeouts for connections. User level timeouts activated.

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 Section 3.7.3, 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 either due to the widespread prevalence of strong correlations and "heavy tailed" data transfers in Internet traffic [FGW98, WTSW97], or due to unusual situations such as flooding attacks, worm propagation, or massively misbehaving software (once we observed one of our local hosts generating 100s of thousands of connection requests; a user was testing a new P2P client). 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

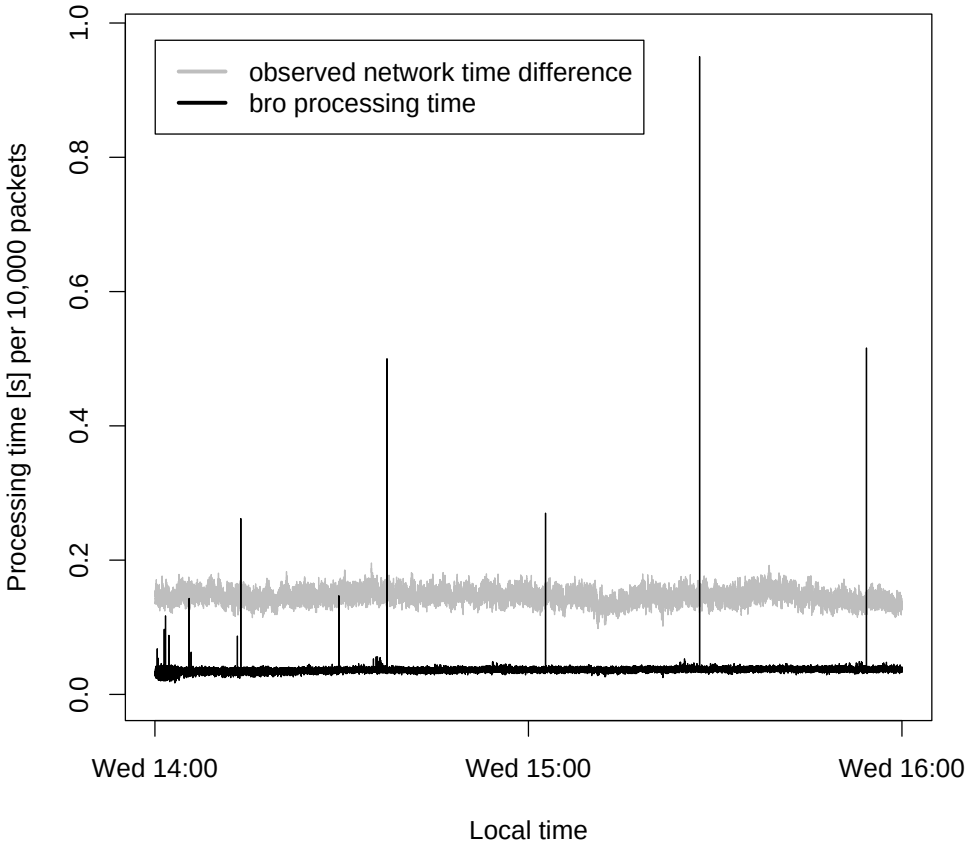


Figure 3.5: Processing time on `mwn-all-hdr`, old hash table resizing spikes

next-to-impossible unless one keeps extremely large capacity margins. As perfect tuning is out of range within the tradeoff 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 [Pax99] 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).

3.6.4 Packet Drops due to Processing Spikes

A NIDS processing traffic in real-time has a limited per-packet processing budget. If this NIDS 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 there are two major reasons for fluctuating packet processing times:

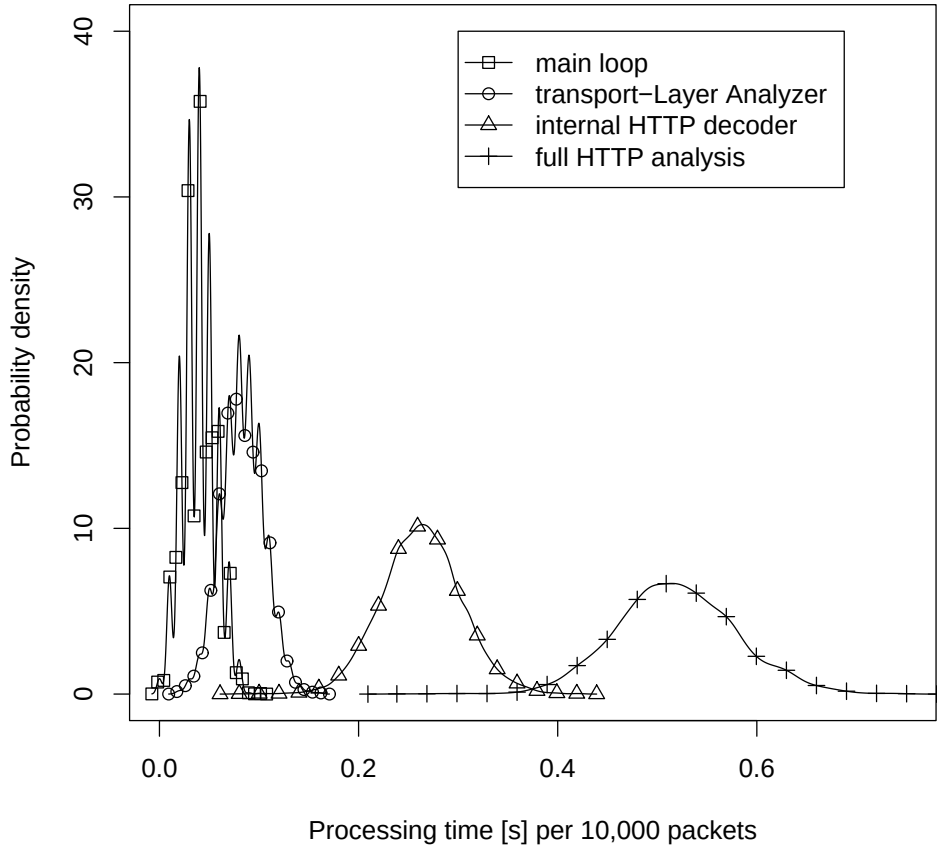


Figure 3.6: Processing time on `mwn-www-full`, fluctuations in per-packet time

First, a single packet can trigger a special, expensive type of processing. For example, 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 3.5 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 Section 3.5 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, we 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 analysis and therefore different processing times. For example, analyzing TCP control packets requires less time than the analysis of HTTP data packets. Yet since the content of packets differs even at the same processing levels, the times can vary significantly. Figure 3.6 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. Section 3.6.3) 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 implies that judging NIDSs in simple terms such as maximum throughput (see, e.g., [HW02]) is questionable.

3.6.5 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. We have repeatedly encountered two kinds of mistakes that inevitably lead to significant problems no matter how minor they may first appear: *(i)* memory leaks, and *(ii)* invalid assumptions about network data.

Even the smallest memory leak can drive the system to memory exhaustion. Simply put, 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 on small captured traces. Yet on large traces, conventional memory checkers like mpatrol [MPa] and valgrind [Val] are terribly slow. In fact, such difficulties originally motivated us to instrument the NIDS to account for its memory consumption, as discussed in Section 3.4. (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 Section 3.6.4.)

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 high rate, as discussed earlier. 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 Paxson's observations of "crud" in network traffic [Pax99] as well as "crash" attacks: 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 they 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 [SM04].

3.6.6 Tradeoff: Resources vs. Detection Rate

So far, we have seen several indications of a rather unusual tradeoff 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 tradeoff 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 already said in Chapter 2, this is a variant of Kerkhoff's principle: while the detection mechanisms are public as the software is open source, their parameterizations are not. We note that choosing appropriate timeouts is not easy. In Chapter 4 we are developing a tool for suggesting reasonable values for a particular environment based on traffic samples.

3.6.7 Artifacts of the Monitoring Environment

So far we have examined problems originating in the NIDS itself. We find that, in addition, high-volume environments also stress the monitor environment (i.e., the monitoring router and the NIDS's network interface card).

First of all, there are some general capacity limitations. At UCB, we monitor the traffic of several routers simultaneously 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 uni-directional 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).

To address this problem, we have developed both a kernel mechanism ("BPF bonding") and user-level support in Bro for merging multiple packet streams. The latter is useful for systems for which the kernel modification is not available. However, even this does not fully address the problem. *Interrupt coalescence* [AGJT03] provides a way to minimize the interrupt load incurred when capturing high-volume packet streams. When coupled with merging multiple streams, however, this can result in the NIDS receiving packets with non-monotone timestamps. Processing them out of order can then lead to incorrect state tracking. To overcome this problem, we implemented a "packet sorting" buffer in which Bro keeps recently received packets for some (user-configurable) time. Now packets with earlier timestamps can then be processed prior to those with later timestamps yet received earlier.

The MWN-router exhibits another strange behavior: it randomly duplicates a fraction of the packets. The only difference between the two exemplars is a decreased hop-count. This suggests that the router puts the affected packets on the monitoring port at two different times: once when they arrive on an input port and another time when they depart on an output port. A support-call has not yet produced any explanation for this

behavior.

The Gbps NIC in the MWN monitoring system is a Intel Pro/1000 MF-LX. To avoid packet losses, we patched the FreeBSD kernel to increase the NIC driver's internal receive buffers and the packet capture library to increase its buffer by three orders of magnitude (after configuring the kernel to allow this).

3.7 High-Volume IDS Extensions

Based on our experiences discussed in Section 3.6, 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.

3.7.1 Connection state management

One major contributor to the NIDS's memory requirements is the connection state (see 3.6). 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 Section 3.6.1 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, it might incorrectly assume that a connection is inactive. Missed packets can be related to drops due to monitoring issues (see Sections 3.6.3, 3.6.4 and 3.6.7). 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/RST 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. We also added the capability that the user's policy scripts can define individual 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 3.2 (bottom) shows Bro's resource consumptions on `mwn-week-hdr` with an overall TCP inactivity timeout of 30 minutes. Note, that the inactivity timer in this example degrades to

a static maximum connection lifetime since `mwn-week-hdr` only contains TCP control packets. In contrast to Figure 3.1 (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 3.2 (top)).

Naturally, inactivity timeouts should be as large as possible. But using timeouts on 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%. Figure 3.2(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 Section 3.6.1 the minimum size of a connection state entry is 240 bytes. To reduce the memory requirements for such connections, we implemented a *connection compressor* to compress their state, leveraging the prevalence of unanswered connection attempts. 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 the 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 in time, connection timeouts are extremely simple. 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[3] for `mwn-`

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

`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.

Two additional optimizations are possible. First, if the responder answers with a RST to a connection request, the connection could be deleted immediately, rather than instantiated as is currently done. In this case, the compressor could avoid instantiating full connection state by directly reporting the rejected connection and flushing the minimal state record. This should be particularly helpful during floods. Second, we can 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.

3.7.2 User-level state management

As discussed in Section 3.6.2, a NIDS may provide the user with the capability to dynamically create their own custom state. In this regard, to cope with the requirements of our high-volume environments, we extended Bro's explicit state management and introduced 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 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,

packets.

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 3.4(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 function simply adjusts the counter and thus maintains consistency between the two tables.

3.7.3 Dynamically controlling packet load

As discussed in Section 3.6.3, 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 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 the 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. (As FreeBSD's packet filter flushes its captured-traffic buffers when installing a new filter, we also had to devise a patch for the kernel-level driver to avoid losing packets.)

Figure 3.7 shows an example of load-levels used operationally at MWN. When the CPU load crosses the upper threshold (Figure 3.7, bottom), the current load-level increases, i.e. Bro shifts to a more restrictive filter (Figure 3.7, 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.

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

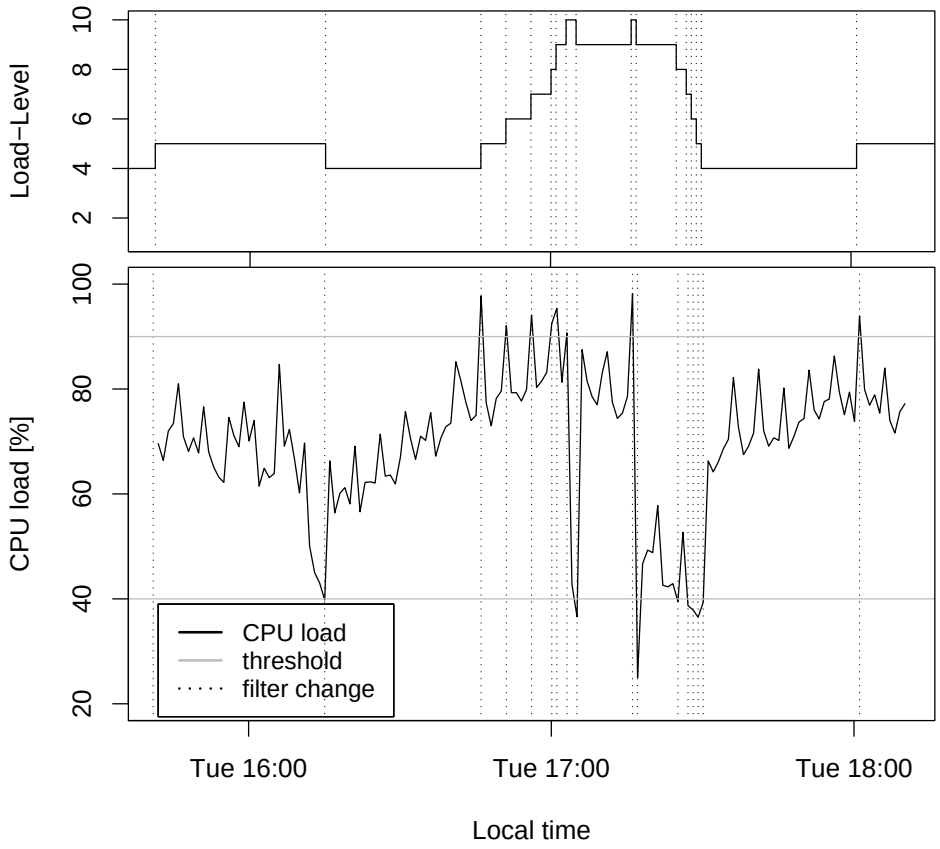


Figure 3.7: Load-levels

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. 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. Note, we are presently working with colleagues on extending BPF to support random and deterministic sampling.

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

3.8 Conclusion

In large-scale environments, network intrusion detection systems face extreme challenges with respect to traffic volume, traffic diversity, and resource management. In this study, we discuss our operational experiences with a NIDS in a Gbps network transferring multiple TBs each day. We identified the main contributors to CPU load and memory usage, understood the tradeoffs involved when tuning the system to alleviate their impact, and devised new mechanisms when existing tuning parameters did not suffice.

Our study is in the context of the Bro NIDS. We are deploying it operationally in a couple of high-performance environments and were faced with several difficulties in terms of memory and CPU exhaustion. While the symptoms often appeared similar, these problems were due to a number of different reasons. First, the 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). Third, even small programming errors (e.g, tiny memory leaks or not fully validated input) will almost certainly bother us eventually. Fourth, independent of the NIDS itself, high-volume traffic also demands a great deal of the rest of the monitoring environment (e.g., the monitoring router and the OS's packet capture subsystem).

For problems that could not be solved with the available tuning parameters, we developed new mechanisms. We improved state management by introducing new timeouts, 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).

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

4 Automatic Resource Assessment for Network Intrusion Detection

4.1 Motivation and Idea

As we have seen in Chapter 3 the performance or more generally the resource consumption of an NIDS greatly depends on the performed analysis and the traffic that has to be analyzed. What kind of analysis a NIDS performs differs from NIDS to NIDS and is configurable by the user. The traffic that has to be analyzed depends on the network's characteristics.

Resource usage is absolutely critical for real-time NIPS and NIDS. The real-time requirement of Intrusion Prevention Systems is to inspect and forward or block packets without delaying benign traffic too much. If the system cannot keep up with the traffic it will eventually drop benign packets due to overload. If the NIPS crashes due to inappropriate memory management, it will effectively hinder legitimate users to access the network. If passive NIDS crash or drop packets, it will not be perceivable by the network users but the detection quality is dramatically decreased.

Concerning CPU usage we treat both, NIPS and NIDS, to be soft-real time systems rather than hard-real time systems: Not-yet analyzed packets can be buffered for some time and a longer than usual delay, occurring sporadically, is not considered to be a critical system failure. Accordingly, we in this thesis do not aim at avoiding any CPU-load spike that is caused by a spontaneous traffic spike. Network traffic is unpredictable enough that this approach would lead to massive over-provisioning of CPU power. In contrast we aim at quantifying what portion of the overall time the system uses more CPU time than available. Thus we are able to react on trends in the network traffic, but ignore sporadic changes in the traffic.

From the NIDS operator perspective it is first of all important to know, how powerful a machine has to be in order to run the desired NIDS. Additionally, during operation, it is extremely helpful for operators to know how much "headroom" there is on the machine running the NIDS: Often the trend in the monitored network bandwidth is known; having an idea of the spare resources allows an operator a guess when new hardware has to be bought in order to keep track with the increased network usage. Lastly if one wants to deploy a NIDS in a new network environment, the available resources constrain the analysis depth that is possible without overloading the system. The operator has to try out different configurations and determine which parameters of the NIDS and the network traffic have significant influence on the NIDS resource usage.

As we already demonstrated in Chapter 3, there are two major dependencies for the CPU usage of a NIDS: *(i)* it rises with increasing network bandwidth that has to be

monitored and *(ii)* it rises the more complex the analysis performed on the traffic is. For memory usage, in Chapter 3 we show the dependency that Bro will use memory resources that scale with the number of connections to be processed. The reason is, that Bro is a stateful connection-oriented system, meaning it associates a certain amount of state with each processed connection.

A NIDS operator has to put a lot of effort into configuring and tuning the NIDS to limit its resource usage. This is difficult, since NIDS provide a large number of configuration options and tuning parameters to adapt the analysis to the needs and the resources in a given deployment. Examples of such parameters in Bro are the *inactivity timeout* (compare 3.7) and whether to analyze a certain application layer protocol. Furthermore, it is often possible to partition the "analysis space". One can use a distributed NIDS and let each "entity" analyze only a part of the total network traffic.

Looking at the parameter set of the Bro NIDS, we find, that there are almost 100 parameters that influence the analysis of the Bro event engine. More than 100 policy scripts included in the Bro distribution manipulate subsets of these parameters, add additional analysis algorithms or simply load other scripts. Although not all of the parameters and scripts have a large influence on performance, it is impossible due to complexity to just try all possible combinations. This means we have to identify a subset of parameters that have a large influence on resource usage.

Finding a configuration for the remaining parameters is still challenging: In order to adapt the NIDS analysis to a network environment, one first has to understand and quantify the impact of the single parameters on the resource usage of the NIDS. This is not as easy as it may look at the first glance: It is heavily dependent on the traffic monitored by the NIDS and thus on the network environment the NIDS is deployed in. It is a well-established property of network traffic, that it is self-similar [LTWW93] meaning it shows high variability both on shorter timescales and on longer ones. This has two major implications.

First if we want to understand when the NIDS overloads (that is when it needs too many CPU cycles) it is not sufficient to look at the average CPU load over, e.g., a day. After all, we consider the NIDS to be soft real-time systems: The buffer has only a limited size, thus in medium to large networks it cannot hold traffic for a whole day. The system simply has to be able to analyze the traffic faster than it comes in. In order to recognize at what point in time the NIDS is not able to keep up with the traffic rate, we have to record and evaluate its resource consumption at high frequency, e.g., every second. Second, if we can model the NIDS' resource consumption in dependence of different traffic parameters and we know how these parameters change over time, we are able to predict the resource usage for the long-term variability.

We want to understand how to assess the resource requirements of complex NIDS. In the literature we find studies that focus on building performance models of various applications. Often such performance models are extracted from the application by decomposing the application into smaller parts: For each component the resource usage is measured. Finally one adds together the resource usages of the single components as they are used for a given input.

Similar to this approach, we show how to decompose the analysis work a NIDS per-

forms into components for which we determine how traffic changes and parameter values influence the resource usage of these components. More specifically, we examine how the resource usage scales depending on traffic characteristics for different values of the NIDS' parameters. We find that for many of these components the workload scales linearly with the number of processed connections.

After having acquired the model of how parameters and the variability in the network traffic influence the NIDS' resource usage, we tackle the problem of how to automatically determine an optimal configuration for Bro even for an, up to then, unknown network environment. The resulting tool assists an operator by systematically comparing different configurations of the system in his network environment. Thus he is able to determine a configuration that provides the best use of the resources at hand: The machine on which the NIDS is running should not or at least only sporadically be overloaded in order to ensure good detection quality. That is: Given a PC system S, a network tapping point P and a NIDS B, can we come up with a set of measurements using S, P and B that help to generate a configuration for B so that B uses the resources of S in a "'optimal" way for its data analysis at P?

We base our tool on resource usage measurements for the analysis of network traces from P. Especially in large network environments recording full packet traces is often not feasible due to resource constraints. Therefore, we show, that Bro's resource usage can be extrapolated from traces that are randomly sampled on a per-connection basis. We find that this technique helps us to drastically reduce the resource requirements for the measurements we perform to test the influence of Bro's analysis parameters.

The rest of this chapter is organized as follows: In Section 4.2 we discuss related literature on performance prediction in different contexts and the use of data sampling for performance measurements. As we later rely on CPU time measurements, we also summarize background information on the accuracy of UNIX process accounting. In Section 4.3 we base our decomposition of Bro from the resource usage perspective on code analysis. In Section 4.4 we perform measurements to *(i)* verify the validity of the models of the single components and *(ii)* examine the accuracy of our approach using traces that are randomly sampled on a per-connection basis. Section 4.5 discusses how we use our model to build a tool to automatically run a series of measurements on a 20 minutes sample of network traffic. The tool aims at *(i)* giving a quick estimate on what analysis is possible given the resources at hand and *(ii)* producing the base for the extrapolation of resource usage on longer-term data as discussed in Section 4.6. In Section 4.7 we briefly summarize our results.

4.2 Related Work

Literature in the area of intrusion detection rarely focuses on resource consumption of NIDS. For understanding how we can predict the resource usage of complex NIDS, we review the literature with the focus of performance prediction and extrapolation. Furthermore we summarize some studies from different contexts that use randomly sampled input data for performance evaluation. These studies deliver interesting background in-

formation, as in our approach, we use random sampling with our input data too. The group of papers in the area of real-time systems follows a much more formal approach than we do. As most studies in this area they focus on asserting execution deadlines for the applications they examine. Last, for understanding how much we can rely on the accuracy of CPU time measurements, we also survey papers about process accounting issues in UNIX systems.

Regarding literature in the Intrusion Detection area, there are quite a number of studies that focus on IDS detection quality. In those studies, the tradeoff between false positives and false negatives is analyzed. Some studies [LFM+02, JU01, LCT+02] take steps towards analyzing detection quality and detection coverage dependent on the cost of the IDS and of the attacks. Gaffney and Ulvila [JU01] focus on the cost that results from wrong detection of the IDS. They develop a model that allows to find a suitable tradeoff between false positives and false negatives dependent on the cost of each type of failure. In contrast, Lee et. al. [LFM+02, LCT+02] focus on developing and implementing high-level cost models for operating an IDS. Such a model is used to dynamically adapt the configuration of a NIDS to the current load of the system ([LCT+02]). The devised models are supplied with metrics of the benefits of a successful detection and self-adapting metrics of the cost for the detection. We argue in Chapter 3 that such metrics are hard to define for large network environments. For adapting the cost metrics, the performance of their prototype systems (Bro and Snort) is monitored using a coarse grained instrumentation of packet counts per second. As we have seen in Chapter 3 this is a oversimplification of complex NIDS. While their basic idea of adapting NIDS configurations to system load is similar to ours, we focus on predicting resource usage of the NIDS depending on the network traffic and the configuration.

In the area of performance prediction and extrapolation, we find three different categories. The first category focuses on the area of supercomputing and targets at predicting a tool's performance on different hardware platforms. A second category uses performance measurements to distribute program components to different machines in a cluster. The third category does not primarily focus on predicting the performance of the applications but on the characteristics and the prediction of the load of machines. Those studies are related to ours in the sense that we use similar techniques for program decomposition and for runtime extrapolation.

The studies in the first category, [MMC04, LHS98, HK98, Men93, MMS95, KM95, Tol95, KUE+00, WF99, SBS96], come from the area of supercomputing and aim at extrapolating the overall runtime of scientific tools or programs on different supercomputers. The focus is on automatically generating models for the programs that are suitable to predict their performance on different hardware architectures and for different problem sizes. This means, the models have to be designed to accept architecture and problem size as parameters. There are different approaches for decomposing applications into components used for modeling: Marin et. al. [MMC04] instrument the code of the application to measure metrics like memory access pattern and how many times a block is executed. The instrumentation is used to determine how the application

scales with different input data. In other studies ([HK98, MMS95] and [KM95]) information gained from the application source code is used to extrapolate the runtime of the program on different target architectures. Mendes et.al. [Men93] are using program traces and scale the time differences between single steps for performance extrapolation but regard different input datasets as open problem. Different data input sizes or problem sizes are in the focus of three other studies ([LHS98, KUE+00, Tol95]): Landrum et. al. [LHS98] use a black box view on the applications. They measure the runtime of scientific computational intensive applications for different problem sizes and produce a polynomial model of the algorithm's runtime. They state the problem with the black box approach to be that they cannot model variation that originates from different data processing paths inside the application. Toledo decomposes a given program into atomic components for which performance can be predicted [Tol95]. The prediction is done using "Benchmaps" that are produced by running the atomic components with different input sizes and interpolate the runtime for input sizes in between. A similar approach is taken by Saavedra-Barrera [SBS96]: The authors split benchmark programs into abstract operations and measure the performance of these abstract operations on the target machine. The sum of all abstract operations is used to predict the runtime of the benchmark program. Another approach for predicting performance depending on input data is simulation with application emulators. Kurc et al. [KUE+00] use emulators in order to be able to only simulate input data, without using actual input. The simulator *Petasim* [WF99] simulates the performance of different memory hierarchies given a user-specified program workflow, description of the target hardware and the distribution of the input dataset.

The second category uses similar techniques but in a different context: These studies ([DCA+03, SS05, UPS+05]) aim at distributing heterogeneous applications like online services to different hosts of a cluster. Unlike the scientific tools examined in the area of supercomputing, these applications are easy to distribute as they are composed of communicating autonomous components like web servers, database engine and application servers. The performance models that are developed for these applications is consequently geared to tweak the tradeoff between computation and communication. The goal of these studies is to minimize the user perceived delay and maximize the throughput of the service.

For the third category the focus is more on modeling host load in a cluster context. Dinda examines the statistical properties of host load using fine grained measurements (1Hz frequency) of the load on many machines [Din98]. Since the background load on a machine greatly influences the running time of jobs to be scheduled, they aim at predicting the host load. Although they discover host load to show self-similar behavior they show that linear models are sufficient for short time predictions of host load [DO00]. Later, a system is introduced [Din01] which determines for a given job and the runtime of that job on an otherwise idle machine the runtime of the same job on a shared machine. Thus, the job can be scheduled on the "optimal" machine. Finding the optimal machine uses the host load prediction techniques of the earlier work. Wolski et. al. [WSH00] present a similar approach to the one of Dinda et. al. [DO00]. The authors develop and use a tool called "The network weather service" and produce comparable results to

Dinda et al. There are also a few papers [KFB99, DI89, SBC06] that focus on predicting the runtime of jobs by analyzing historical data: They build a knowledge base over time for different problem sizes and parameters and predict the runtime of a new instance by interpolation or data-clustering techniques.

We have already seen quite a number of studies in the supercomputing context that focused on different input sizes. However none of them used random sampling of the input data in order to reduce the time needed for the measurements. Lang and Singh [LS00] use random sampled query datasets for evaluating the performance of database index structures. One of their results are formulas to extrapolate the page layouts from the index structures resulting from the sampled input data. There is another study that analyzes problems of sampled input data for reducing time needed for experiments [CB98]. The authors use sampled memory access traces to assess memory system performance, i.e. cache performance. Sampling of the input trace in this context works by omitting sequences of accesses from the trace (block-wise random sampling). The problem that arises is that at the beginning of each sample, the state of the simulated cache is unknown. They evaluate different methods for re-initializing the cache content at the beginning of each sample. The "stitch" strategy, which assumes, that the cache at the beginning of a sample has the same state as at the end of the previous works best for accurately (within 90% confidence intervals) simulating the cache miss ratio for caches up to 64KB in size.

Literature focusing on real time systems ([Sha89, CBW96]) targets formal guaranties on worst case run times. Although this is a requirement for hard real-time applications, the NIDS we are focusing at are too complex to apply these strict formalisms. But in this context there are also studies, that focus on the decomposition of program code: Facchini et. al. [Fac96] use a similar approach as Saavedra-Barrera et. al. [SBS96] to decompose real-time programs into basic operations in order to predict their runtime in early design stages.

In terms of process accounting accuracy of operating systems, McCanne and Torek [MT93] discuss that process accounting in UNIX systems may be flawed, if periodic sampling is used to account the CPU time to processes. In case a process is somehow exactly synchronized to the accounting clock, the process may for an extremely synchronized process, be charged always or never. The authors implement random intervals for sampling the CPU usage. Their solution is integrated in systems later than version 4.4BSD. Etsion et al. [ETF03] evaluate the benefits and the cost of using higher than default clock interrupt rates. Although their focus is primarily on reducing delay on soft real time multimedia applications they conclude that higher clock interrupt rates significantly increase the accuracy of process accounting while introducing negligible overhead.

4.3 Understanding the Resource Usage of Bro

For this work our goal is to model the Bro NIDS in a way that we can automatically determine a working parameter set for Bro in the given network environment. Therefore we need a good understanding of Bro's resource requirements and of the dependencies between resource consumption and analysis. To achieve this, we leverage our findings from Chapter 3 and perform a detailed code analysis. We decompose Bro's analysis along Bro's structure into the contributing components. For each component, we determine the influence of network traffic variations. To determine how changes of the traffic composition impact overall resource usage, we can then extrapolate the resource usage of the individual components and piece them together to the overall resource usage. The three components we distinguish are *(i)* basic connection handling, *(ii)* Bro analyzers and *(iii)* the signature engine. We now discuss the contributions and the dependencies of each component in turn.

4.3.1 Basic Connection Handling

Since all analysis in Bro is connection oriented, we consider the component "basic connection handling" as the common baseline for any further analysis Bro is able to perform. In the following we term a Bro configuration that does only basic connection handling BroBase. Remember that Bro relies on BPF packet filtering in order to minimize its workload. For running the BroBase configuration, Bro analyzes the connection control packets only, that is all packets with any of the TCP flags SYN, FIN or RST set.

The workload of the analysis done with the BroBase configuration is composed of the workload contributed by packet handling and work done for event interpretation. The term "packet handling" pools a number of different tasks, e.g., IP checksum verification, transport layer protocol analysis and state update, e.g., update of the TCP sequence number. Event interpretation consists in the BroBase configuration mainly of logging, i.e., the logging of connection summaries. The CPU time needed for performing the basic packet handling obviously directly depends on the number of processed packets. The transport layer protocol analysis for TCP should be largely dominated by the instantiation and expiration (that is also timer management) of connection state (see Section 3.7.1). After all if the packet filter described above is in effect, almost all analyzed packets trigger the instantiation or the expiration of connection state.

The last contributor to the workload for basic connection handling is the interpretation of the events by the policy scripts. Our minimal configuration BroBase only generates one line connection summaries; meaning it generates log entries by evaluating only a few (i.e., a fixed number of) events per connection. Therefore we can expect, that the CPU time used overall scales linearly with the number of processed connections.

The connection compressor outlined in Section 3.7.1 introduces an important exception. This extension causes some connections to be treated rather different: If a connection request is not answered, full connection state is never instantiated. Therefore we have to distinguish between the workload contributed by the state management for the full-instantiated connections, and the workload contributed by the connection com-

pressor for handling the unsuccessful connections. As our experiments in Section 3.7.1 indicate, the overall workload is usually dominated by the fully instantiated connections. The smaller workload contributed by the connection compressor scales linearly with the number of handled unsuccessful connections.

Looking at the memory consumption of basic connection handling, we learned from Section 3.6.1 that there is quite a bit of state associated with each connection inside the Bro transport layer analysis component. Besides this "core state", Bro also keeps "user state" (see Section 3.6.2). User state is bound to variables and data structures defined and maintained by the user in Bro policy scripts. For the configuration we are talking about here, minimal connection handling, there is no per connection user state. This implies, and our results in Section 3.6.1 indicate, that Bro's memory usage for the connection handling is expected to scale linearly with the number of connections processed: All data structures involved in connection handling in Bro have usually roughly the same size for each connection they are associated to. Again, with the use of the connection compressor we have to distinguish two cases: Successfully established connections consume more memory than unsuccessful connection attempts. Within these categories, however our observation remains the same: Bro holds a fixed amount of state for each established connection and another fixed amount for each connection attempt.

4.3.2 Bro Analyzers

Bro's connection handling provides a basis for a variety of analyzers. Remember, that a Bro analyzer may consist of two parts: The first is hard-coded in Bro's event-engine, parses the network traffic and digests it into an abstract and policy neutral description. The output of this "*parser*" are analyzer-specific events. The second part is a user defined interpreted script that uses the information extracted by the parser to make site or policy specific decisions on whether the observed behavior is benign or malicious.

Many analyzers that are by default shipped with Bro, especially the application layer protocol analyzers consist of both parts. For the application layer protocol analyzers, the parsers in Bro's event engine decode the payload stream according to the respective protocol and extract the contents into appropriate data structures. These are used by the policy script part of the analyzer to enable user defined policy decisions.

There does not need to be a direct mapping between a parser in the Bro core and a single policy script: It is possible to write policy scripts (which resemble user-defined analyzers) that use the events generated by other (or more than one) event engine parsers. An example for this is the scan analyzer-script included in Bro's standard distribution. It uses the events extracted by the basic connection analysis (i.e., the TCP, UDP and ICMP parsers) to maintain its state about scanners at the user level. It is also possible to define policy scripts that rely on events or data structures of other policy script analyzers. The author of such policy scripts has to make sure that the other analyzer(s) a script depends upon are loaded as well[1].

[1] Bro's type checking will assist the author: e.g., if he redefines a variable but has not loaded the corresponding base script, the interpretation of the policy script will fail with an error

Implications on Resource Usage

In contrast to our basic connection analysis, each analyzer contributes some analysis work done for only a subset of all packets processed by the whole system. For example, if Bro is configured to use its FTP analyzer, the actual work of parsing FTP commands and replies is done only for the fraction of packets that belong to FTP connections. On the other hand the low-level BPF filtering leads to a dependency between the application layer protocol analyzers and the basic packet handling: As soon as additional application layer analyzers are loaded, the filter has to be adapted to allow more packets to pass, i.e. the packets with the payload to analyze. These additional packets do of course add workload onto the components in the basic connection handling. Especially the per packet operations (e.g., checksum verification) have to be performed for each additional packet and also the connection state is updated for each of these packets which involves a connection lookup each time.

As mentioned above, for memory consumption we distinguish between core state and user controlled state. These two categories correspond directly to the two parts each Bro analyzer consists of. The "core state" is the state needed by the protocol decoder to reconstruct the endpoints' state. It is directly associated to Bro's per connection state. For example the HTTP analyzer uses its additional connection state to separately decode the single HTTP requests and replies in persistent HTTP connections. The "user state" is state that is controlled by the user on Bro's policy script level. The policy scripts may define and use arbitrary large data structures such as tables, sets or vectors. An example for user-controlled state are the tables in Bro's scan analyzer: These tables are used to keep track of host addresses that contact too many other hosts within a certain timespan.

Decomposition of Analyzer Workload

For the basic connection handling we are able to derive our expectation on how resource usage scales directly from an analysis of the Bro program code. If we are looking at Bro's analyzers we make the following observations:

1. Each application layer protocol analyzer performs its analysis independent of other loaded application layer protocol analyzers

2. Since the user implements his site's policy with the powerful Bro scripting language, it is in general impossible to derive the exact amount of work the analyzer does from only looking at the program code.

The first observation implies, that the amount of work an analyzer performs is the same, no matter what other analyzers are concurrently activated. This allows us to assess the CPU time needed and the memory consumption for each single analyzer: We can just run Bro only performing basic connection handling and measure its used CPU time and memory consumption as a baseline. After that we run Bro with each individual analyzer that has no dependencies besides basic connection handling. The difference in CPU time is contributed by *(i)* the additional analyzer processing "its" connections in

the given trace and by *(ii)* additional per packet work for the extra packets, that pass the BPF filter and that the transport layer analyzer additionally has to analyze.

Accordingly, the additional memory consumption has to be due to the additional state the analyzer introduces for "its" connections. The independence of the single analyzers also means, that we can expect the CPU time and memory consumption of these analyzers to be additive as the following example illustrates: First, we run the BroBase configuration and determine the CPU time usage and the memory consumption on a trace. In a second step, we run the BroBase configuration plus analyzer A, which needs, e.g., an additional 10 sec. CPU time and 1 MB additional memory for the same trace compared to the instance running only the BroBase configuration. Then, we run the BroBase configuration plus analyzer B, which needs, e.g., an additional 20 sec. CPU time and 0.5 MB additional memory on the trace. If the analyzers A and B are additive, a run of Bro with the BroBase configuration and both analyzers (A and B) should need 30 sec. additional CPU time and 1.5 MB additional memory compared to running the BroBase configuration alone.

The second observation implies that we cannot make a general statement on how the work done by the individual analyzers scales. Looking at the code of the application layer protocol analyzers inside Bro with the task of decoding and parsing the different protocol, we expect the CPU-load in the worst case to scale linearly with the number of bytes processed. This is based on the fact that all data structures that are used within the protocol parsers in Bro should have a worst-case constant running time (compare 3.6.4) per packet (each with a fixed maximum amount of payload). Therefore, in the worst case the distribution of connection sizes determines the overall cost of CPU time for a specific analyzer. The same holds for the default policy script part of the analyzers: excessive processing spikes triggered by individual packets have to be avoided in order to not trigger packet losses. Therefore Bro's default policy scripts are designed so that they have a constant worst case runtime per event. In the worst case, each packet with payload triggers an event (technically, one packet can trigger multiple events; remember however, that Bro's design idea is, that the event engine reduces the data stream by generating less events than it analyzes packets). Therefore the workload added by the scripted part of the analyzers should in the worst case scale with the number of processed packets or bytes.

In practice, analyzers are sometimes programmed to skip certain parts of the payload, since those parts are not analyzed. For some analyzers and their default policy scripts it is even apparent, that the work done by the analyzer is dominated by the work done at connection setup and does not depend on the number of packets exchanged after that phase. An extreme example is the SSL analyzer. It only parses the protocol and generate events for the setup phase of the connection, where certificates are exchanged. After that phase, the actual data exchange does not trigger any more events, since there is no point in doing protocol analysis for encrypted connections. That means that the overall work done by the application layer protocol analyzers is either dependent on the distribution of connection size (in bytes transferred) or on the number of connections that use the protocol.

Quite a few application layer protocol analyzers extend Bro's connection state in the decoder part of the analyzer. Usually this is a fixed amount of additionally allocated memory per connection. This implies, that the memory consumption for this additional state scales with the number of connections in state which are parsed by that analyzer. For the memory consumption of the script level part of the analyzers we have the same fundamental problem as for CPU time: As the user should specify his own policy scripts we cannot necessarily assume a certain behavior of the analyzer in terms of memory consumption. Basically the user could come up with a script that consumes memory unrelated to the network traffic (e.g., triggered by time). In such a case our approach of associating resource usage with traffic characteristics will not work. However for scripts that associate script level data structures to each connection, memory usage will scale with the number of connections that are concurrently kept in state.

Looking at the default policy scripts included in Bro's distribution, there are analyzers that associate state to a set of connections (not to every single connection). An example is the default configuration of Bro's HTTP analyzer: For every new *session* it instantiates a rather large user-level data structure. A session according to the HTTP analyzer's semantic consists of all connections from a HTTP client to one HTTP server within a certain time interval. The data structure associated to the session does not grow when new connections belonging to the same session are added. In this example, the user level memory usage does not scale linear with the number of connections. Here it scales linear with the number of analyzer-defined sessions.

4.3.3 Signature Engine

The signature engine [SP03b] in Bro is an add-on in the sense that much of the analysis Bro performs does not rely on signatures. The signature matching engine can match packet headers, raw payload streams and application layer protocol specific payload streams (as extracted by the corresponding analyzer) against predefined byte patterns. Payload stream matching in Bro is done using a finite automaton (DFA) implementing a regular expression matcher. The compute time for Bro's regular expression matching scales always linearly with the input length of the stream. Since each stream is divided into packets with a limited size, the time needed to match the content of a packet has a fixed upper bound. Thus signature matching in Bro can be considered to be a special analyzer, that adds CPU time to the "basic packet handling" category. In its default configuration the signature engine performs its matching only on those packets that pass the low-level packet filter. We described before, that the packet filter depends on what analyzers are loaded. Thus additional analyzers add workload for the signature matching as they supply it with more packets.

The memory usage of the signature engine depends primarily on the size of the DFA for the regular expression matching. Depending on the size and the complexity of the signature set to match, these DFAs can grow very large. Due to the dynamic construction of the finite automata for regular expression matching, the memory usage for the DFA is effectively limited. However, dynamically constructing the DFA influences again the CPU usage: In the worst case, each byte that has to be matched implies the construction

of a new state in the DFA. A detailed empirical evaluation of the performance of Bro's signature engine can be found in [SP03b].

4.4 Bro Resource-Usage Measurements

After our description of Bro's resource consumption from code analysis we now present results from actual resource consumption measurements of the Bro system. The focus is on understanding and validating the influence of different configurations of Bro and different traffic samples. The primary goal of this work is to find a configuration for Bro that does not overload the machine, thus avoiding packet drops. We start by analyzing the interactions between machine load, processing time and packet drops. Then we discuss a sample resource usage "footprint" of a Bro instance running the BroBase configuration on live traffic and compare it to the resource usage of a Bro instance running on the same traffic, this time reading it from a recorded trace file. After this we systematically verify our findings from Section 4.3: First we validate, that the workload and additional state introduced by the different application layer protocol analyzers is additive. Then we use traces that are randomly sampled on a per-connection basis to analyze, how Bro's CPU usage and memory usage scales with the number of connections processed.

4.4.1 Setup and Datasets

The network environment we base our measurements in is the Münchener Wissenschaftsnetz (*MWN*), Germany. The setup of our monitoring systems can be seen in Figure 2.4. Note, that this setup allows us to analyze and capture live network traffic in parallel on different machines.

For all measurements done in this context we use development release 1.1 of the Bro NIDS running on one of our AMD Dual Opteron machine with FreeBSD 6.1 as operating system. The Bro version in use includes the built-in instrumentation presented in 3.4 and 3.5 plus some extensions: For our measurements we configure the instrumentation so that CPU time information is output for every network second. Additionally to CPU time spent per second we output the additional amount of memory allocated during the last second. While this does not give us an exact breakdown of how much memory the individual internal data structures consume, it does also not add processing spikes due to recursively scanning through all data structures. If Bro runs on live traffic our instrumentation also outputs the number of dropped packets for processing each elapsed second of network traffic.

For our systematic measurements of Bro's resource consumption, we captured a 24-hour *full* trace at MWN's border router on October 11, 2005. As this is hardly possible with commodity hardware, we used a high-performance Endace DAG capturing card [End]. The trace encompasses 3.2 TB of data in 6.3 billion packets and contains 137 million distinct connections. 76% of all packets are TCP. The DAG card did not report any packet losses. In the rest of this thesis we refer to that dataset as `mwn-full-packets`.

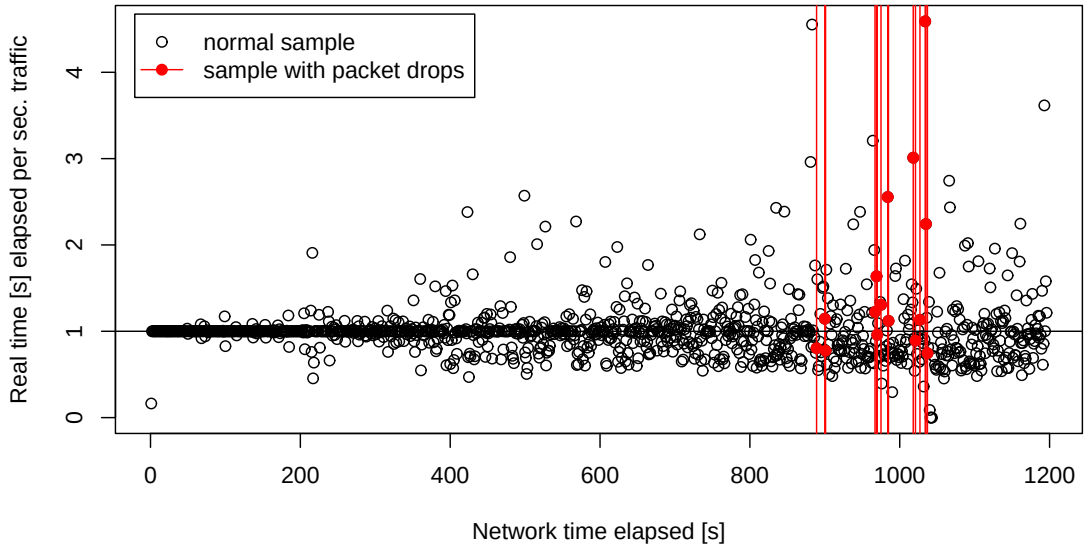


Figure 4.1: Relation between elapsed real time and packet drops

4.4.2 Understanding Packet Drops

In general, packet drops have to be avoided in security monitoring systems such as NIDS, since non analyzed traffic may imply attacks not detected. In a real time NIDS like Bro packet drops occur when the system is overloaded. Using our instrumentation, we can correlate packet drops with the load that the Bro process imposes on the system. Figure 4.1 shows for each second of a 20 minute run of Bro on live network traffic (x axis) the real time that has elapsed after Bro has processed this second of network traffic (y axis). The Bro process is started at time 0. The red vertical lines denote the seconds after start in which packet drops were reported by libpcap.

For a run where no problems occur, we expect the measurements of "real time elapsed" to be always 1 sec.: Bro has ample of CPU headroom, and traffic is processed faster than it comes in. Packet drops should occur when the elapsed real time for processing one second of network traffic significantly exceeds one second. At what time exactly packets are dropped depends on the amount of available buffer space and buffer organization for packet capturing (in our configuration using FreeBSD's capture and filtering mechanism two times 10 MB organized as double buffer) and the volume of incoming traffic.

We note from Figure 4.1 that directly after starting the Bro instance everything is fine: the real time elapsed per second network traffic is exactly one second, meaning that Bro processes the packets faster than they come in. Then at some times more than one second real time has elapsed after Bro processed one second of network traffic. As long as these are single outliers and not too large they are compensated by buffering: For the following second(s) of network traffic less than one second real time elapses as these packets can be read at processing speed from the packet buffer, meaning the process

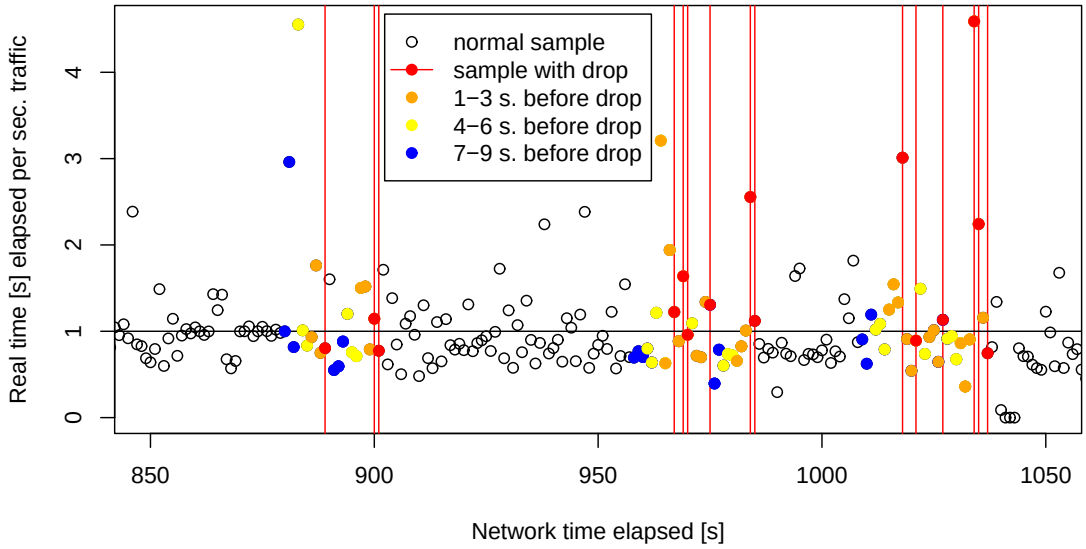


Figure 4.2: Relation between elapsed real time and packet drops, zoom-in

does not need to wait for new packets to arrive. With our 10 MB buffer we observe outlier samples in the order of 2.5s without causing packet drops.

The first packet drop occurs after an excessive spike in elapsed real time of more than 4 sec. Figure 4.2 shows a zoom-in of the Section with the packet drops from Figure 4.1. The nine samples immediately preceding the sample where packet drops occur are colored: sample 1-3 before drop orange, 4-6 before drop yellow and 7-9 before packet drop blue. The first packet loss occurs not immediately during processing the "expensive" sample during which more than 4 seconds real time elapse, but six samples (network seconds) later. This means, that at the time when packets are lost, our buffer is completely full and should contain the coming seconds of network traffic. The number of seconds buffered should correspond to the *lag* that was built up before (e.g., for the first drop ca. 5.5 seconds). However the lag that fits into the buffer depends on the stream of freshly arriving packets.

We note that the elapsed real time is not only influenced by the user and system CPU time accounted to the Bro process. Even if the Bro process runs on an otherwise idle machine (as it was the case for the instance we ran here), depending on the network traffic, the hardware and the OS, the system is more or less busy to handle interrupts and packets (e.g., packet filtering). Since we are using a machine with two processors and an OS suitably configured for SMP (Symmetric Multiprocessing), we observe the elapsed real time as plotted in Figure 4.1 and Figure 4.2 to be roughly the same as the user CPU time accounted to the Bro process (the system CPU time accounted to the Bro process is in fact negligible). Running the Bro process on a machine that is not idle causes the elapsed real time and the user CPU time accounted to the Bro process to differ significantly.

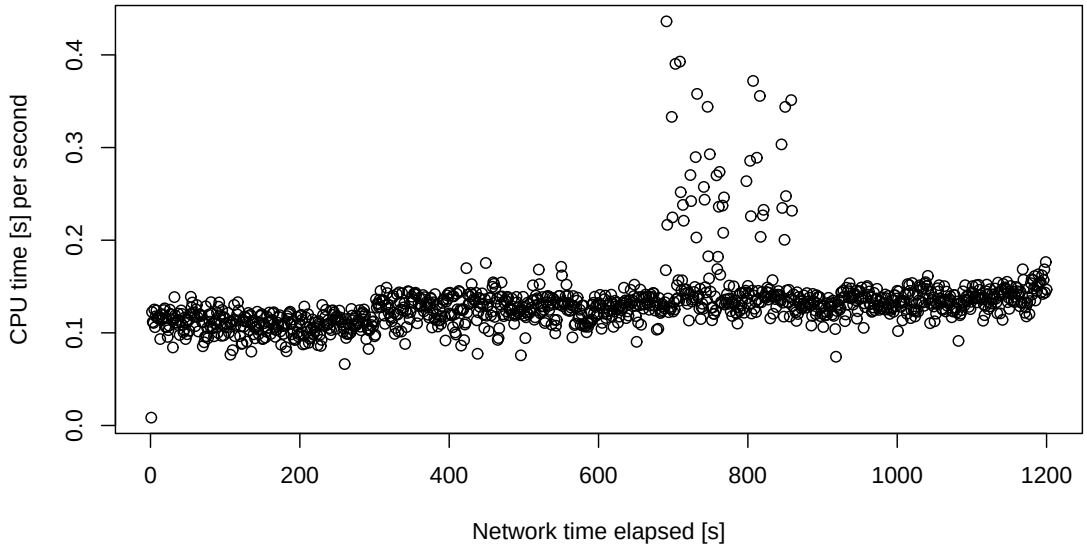


Figure 4.3: CPU usage for Bro instance running on live network traffic

We conclude that the "lag" that builds, when Bro does not process the incoming packets fast enough is closely related to the probability of packet loss. However, even in our setup with an otherwise idle SMP machine it is extremely hard to exactly predict less quantify packet drops while processing live traffic. The main reasons are that *(i)* single outliers are buffered away and that *(ii)* the buffer capacity (in seconds) depends on the traffic volume yet to come. On the other hand it is desirable to keep the "lag" small anyway: Just think of inline IPS and, e.g., interactive traffic - in that context "lag" implies packet delay which is annoying to the users. However we notice, that hardware, OS and scheduling issues may make it hard to infer the "lag" from the accounted user CPU time of the traffic processing system.

4.4.3 Bro on Live Traffic

In the previous example, we have seen, that there is a direct relation between the user CPU time needed by the Bro process and the overload of the system which in turn results in packet losses. Now, we systematically examine how Bro's resource usage changes over time and how its parameters and how traffic characteristics influence the CPU usage.

Figure 4.3 shows an example of a measurement of CPU time per second trace time accounted to the user-level of a Bro process. For this we ran an instance of Bro for 20 minutes on a FreeBSD 6.1 system reading the packets live from the network and only performing basic TCP connection handling (the BroBase configuration). Overall, there are three observations:

1. There is a jump at 300 seconds after starting Bro

2. There are a number of outliers between 700 and 900 seconds after starting Bro
3. Otherwise the fluctuation for our metric user CPU time per elapsed second of traffic is rather small

The jump at 300 sec. occurs since Bro's attempt timer was configured to that time. At 300 sec. after startup the first timers for connections that do only consist of connection attempts, meaning TCP SYN packets, expire. For each of these connection attempts an event is triggered which generates a connection entry for the logfile. After that the connection state is removed from the connection compressor. In general that means, that Bro processes need less CPU time for the first X seconds, where X denotes the value of Bro's attempt timer variable.

For examining the outliers between 700 and 900 seconds after starting Bro, we manually inspected the connection logs of that run. We found that in this time interval larger than usual numbers of connection attempts are expired. Upon looking closely at Figure 4.3, we notice small outliers between 400 sec. and 600 sec. after start. These are caused by surges of TCP SYN packets (ca. 3000 SYNs per second) which the connection compressor handles very well: The attempts do not cause an immediate spike in CPU usage. However, exactly 300 sec. later the timeouts for all the unsuccessful attempts are expired which causes the larger load spikes as for each attempt a connection summary is generated. Although these outliers are specific for that very example of network traffic, it illustrates the general variability of network traffic. Measuring the CPU usage of another Bro instance in the same network environment at a different time will most likely show similar outliers as observed in this example.

If we look more closely at the numbers between 300 sec. and 1200 sec. after start except the outliers, we note that all measurements are in a relatively narrow range from 10 to 15% CPU load (which corresponds to 0.1s CPU time for 1s network traffic). The mean CPU time here is 0.13 seconds per second network traffic and the standard deviation is 0.01s. The remaining fluctuation is caused by the fluctuation in the network traffic itself and the resulting fluctuation of the number of analyzed packets/connections.

This experiment shows, that in general Bro's CPU consumption after a startup phase is fairly stable. The duration of the startup phase is determined by Bro's connection timeouts. If we want to compare the measured CPU time with the CPU time needed for a long running Bro instance, we have to ignore the measurements during the startup phase. Obviously, outliers in the traffic cause outliers in the resource consumption too, but overall we notice that the median and the mean load the Bro process imposes on the system is a reasonable metric for summarizing the resource usage of the BroBase configuration.

4.4.4 Bro Live vs. Bro on Traces

As we plan to use traces for our measurements and in our assessment methodology the first question that comes to mind is: Is the CPU time used by Bro running online on real network traffic the same as for Bro running on a trace? Depending on the mechanism the packets are acquired from the network interface card [Sch05] at least the system itself

may be under greater stress if it has to capture all packets from the network instead of reading them from disk on demand. From the Bro process' perspective there is not much difference whether the packets come directly from the network or from a file. In both cases libpcap [Lib] supplies the process with the next packet after the previous one has been processed. Looking at libpcap there is one major difference in terms of CPU time accounting. If the packets are acquired from the network directly, on most operating systems the BPF filter code is executed in the kernel. In this case it is accounted as system load and not accounted to any user process. On the other hand, if reading a trace from disk, the filtering (using the same BPF code) is done in libpcap itself, therefore being accounted to the process' user time.

Our instrumentation for measuring resource usage in Bro is purely triggered on the time in the network traffic (no matter whether live or reading from a trace file): We save the timestamp of the last packet when a resource usage summary was output and check the network timestamp of each packet analyzed, whether a new summary is due. Thus, we measure CPU time per second network traffic. In a scenario where network traffic is sparse, the difference of running Bro on the live traffic and reading from a trace file becomes apparent: A Bro instance reading its packets from the network would leave the system more or less idle by processing the packets as they come in. A Bro instance running on the captured trace of the same traffic instead, uses all CPU resources it can get in order to process the packets as fast as it can and therefore terminate after a fraction of the timespan the trace covers. In both cases we expect the output of our measurement to be very similar as we account CPU time per network (or trace) time. However, depending on the system it is running on, the instance of Bro working on the captured trace could profit of better caching hits or less scheduling-imposed penalties.

To directly compare an online instance of bro on real traffic with an offline instance we use two separate machines having access to the same network data (see Section 2.3). In addition to the first system running Bro on the live network traffic, we utilize a second system to capture a packet trace of all packets that pass the BPF filter of the live Bro instance in parallel. Afterwards, we copy the recorded packet trace to the machine we ran the online Bro instance on and analyze it with another (offline) Bro instance. Figure 4.4 shows the direct comparison between the live Bro instance already discussed in Figure 4.3 in black circles and the offline instance on the trace in red triangles. We note that indeed the difference is very small. All measurement samples of the online run, including the outliers between 700 sec. and 900 sec. are closely matched by the measurement samples of the offline instance. Overall the offline instance needs a bit less user CPU time than the online instance: The mean CPU time per second is 0.0059 seconds lower for the offline Bro run. We speculate that this is due to system subtleties like better caching or even due to minor CPU time accounting imprecision on the used OS.

So far, we have seen, that we can indeed directly compare the output of our instrumentation for offline instances of Bro with the output for the resource usage a live instance of Bro has when analyzing the same traffic. In further experiments we verified, that live CPU usage is comparable to offline CPU consumption not only for basic connection handling but also more complex configurations. For this we used the same technique as

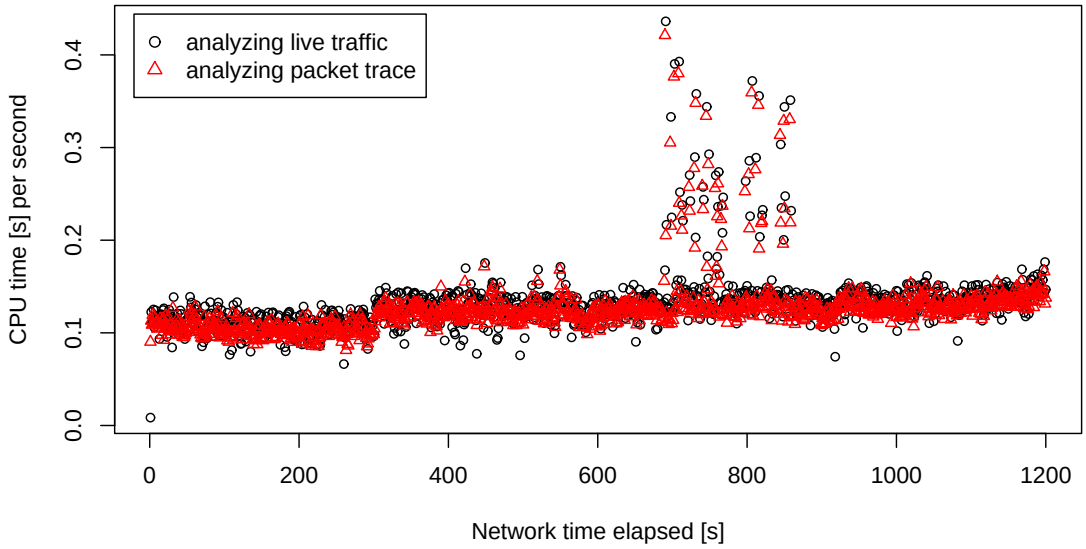


Figure 4.4: CPU usage for Bro instance running on live network traffic vs. recorded packet trace

described before: we run Bro on the live traffic and in parallel capture a trace of the analyzed packets on a different machine. For example, we configured Bro to perform HTTP request analysis in addition to the basic TCP connection handling. Comparable to our result before, in this case, the mean CPU time per second trace is 0.0034 seconds lower in the offline run.

4.4.5 Analyzer Workloads

We now verify our claim, that the application layer protocol analyzers perform their work independently. This enables us to sum up the contributed workload of a number of analyzers and thereby get the total workload for a configuration with all these analyzers loaded. For this we use a 20 minutes excerpt of our `mwn-full-packets` trace on which we run different instances of Bro. The first instance runs with the BroBase configuration. The second instance runs the BroAll configuration: Additionally to the BroBase configuration, BroAll includes the Bro analyzers *login, ident, ftp, finger, portmapper, frag, tftp, http-request, smtp, ssh, pop3, irc,* and *ssl*. 13 more instances run all the additional analyzers contained in the BroAll configuration separately (each in addition to the BroBase configuration).

In contrast to the experiments described before, here we are using a full packet trace, containing all packets that cross the tapped link. Remember, that in the previous experiments we did only capture those packets that get actually analyzed by the activated configuration; For our main example, the BroBase configuration, a rather small subset. Since for live Bro instances the packet filtering is done in the kernel of the OS, the

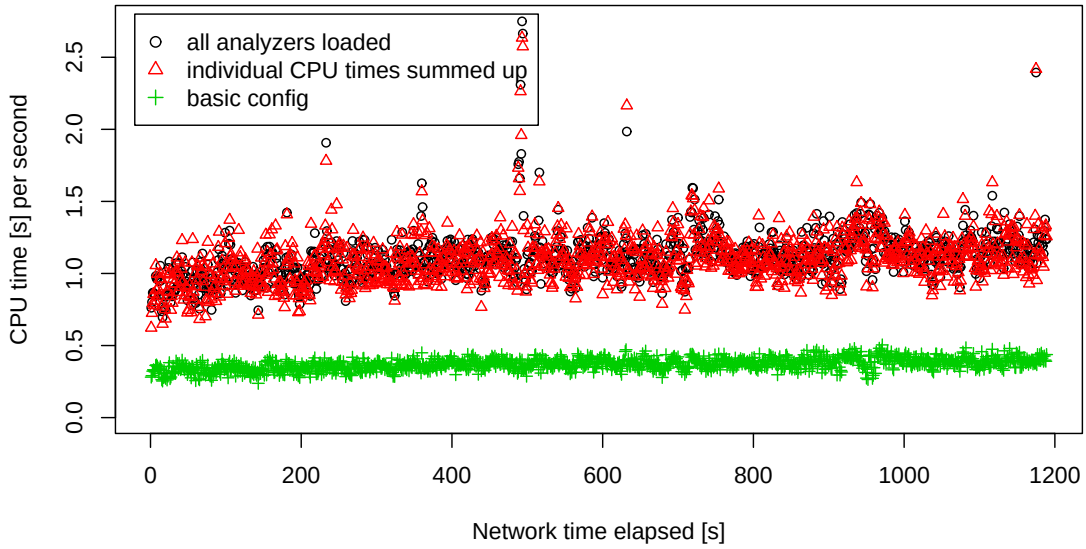


Figure 4.5: Accumulating analyzer CPU usage

workload introduced by packet filtering is not accounted to the Bro process.

Consequently we let our Bro instances not read the raw traces directly but prefilter them to contain only the packets analyzed by Bro. Therefore we used a separate process to prefilter the full packet trace and fed the result via a Unix pipe to the analyzing Bro instance. In our setup it turned out, that data from the full (unfiltered) packet trace could not be read and prefiltered at a sufficient rate to keep the Bro instance busy, which caused waiting times. Seemingly these waiting times significantly interfere with FreeBSD's process accounting resulting in higher CPU usage times. Therefore for all experiments with the `mwn-full-packets` trace, we prefiltered the trace first to disk and ran our Bro instances on the resulting traces.

For each Bro instance running one analyzer additional to the BroBase configuration, we determine the difference in CPU usage between the instance itself and the instance running the BroBase configuration. In Figure 4.5 the black circles denote the user CPU time used by the BroAll configuration. For comparison, the sum of the user CPU time of the BroBase configuration and all additional user CPU times of the single analyzers relative to the BroBase configuration are plotted as red triangles. The green '+' symbols denote the user CPU time of the BroBase configuration itself. Everything above the green '+' symbols is contributed by the additional analyzers. We make two observations: *(i)* If we compare the BroBase configuration to the BroAll configuration we note, that the additional analyzers add quite a bit of fluctuation. Especially the spike at ca. 500 sec. is only contributed by the additional analyzers in contrast to our example before in Figure 4.4 where we had spikes in CPU usage for basic connection handling as well. *(ii)* Accumulating the additional workloads of the single analyzers produces very similar CPU usage numbers as the actual measurements for the BroAll configuration.

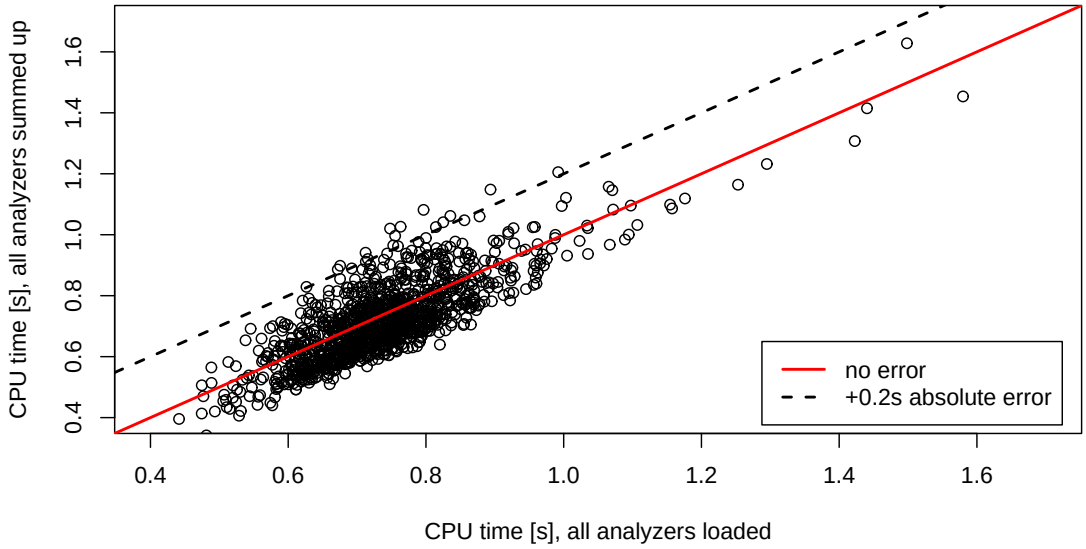


Figure 4.6: Scatter plot accumulated CPU usage vs. measured CPU usage

However, the fluctuation seems to be slightly increased while the summed up workload tends to be a bit smaller than the workload directly measured. In order to get a sense for the variability across different types of analyzers, Appendix A contains separate plots (analog to Figure 4.5) for each of the 13 analyzers included in the BroAll configuration.

The scatter plot in Figure 4.6 allows us to directly compare the additional CPU time accumulated for all analyzers to the CPU time measured with the Bro instance running the BroAll configuration. We omitted all samples greater than 2.0 seconds from the plot to focus on the large majority of samples between 0 and 1.7 seconds. The single outliers that were omitted can be well recognized in the timeseries plot (see Figure 4.5) to match rather well. From Figure 4.6, we notice that there are quite a number of samples where the sum of the CPU times exceeds the measured one by an absolute error of roughly 0.2s (20% CPU load, black line). The mean relative error is 9.2%, the median relative error is 8.6%. Since there are no outliers that do significantly contribute to this relative error, we want to understand its origin.

By looking at the relative contribution of the single analyzers, we find that there is quite a number of analyzers that do not add significantly to the workload. Mostly, these are analyzers that examine connections that are not prevalent in our trace (or in the network environment in general). For MWN that is, e.g., connections analyzed by the Finger analyzer. The workload for the instances running these analyzers is almost the same as for the instance running the basic configuration. Due to the precision of the operating system's resource accounting two measurements of the same workload can never be exactly the same; in fact, when running the same BroBase configuration ten times the mean range of our measurement samples is 0.018 seconds. Note, that this range is much smaller than the workload-induced variation within a run of Bro with

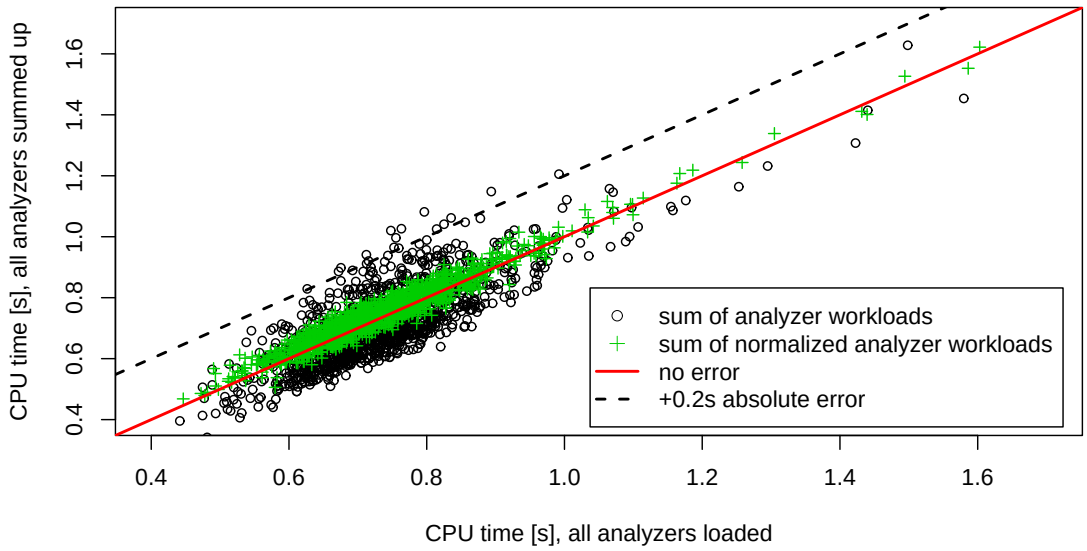


Figure 4.7: Scatter plot normalized accumulated CPU usage vs. measured CPU usage

the BroBase configuration (e.g., see Figure 4.5), which is why we do not add error bars to these plots. For analyzers contributing very little workload, this means that their contribution to CPU time usage to each sample cannot be distinguished from the measurement-induced variation of the sample. The fluctuation of all single runs with one analyzer activated each, may well accumulate to the total variation seen in Figure 4.6.

Figure 4.7 is identical to Figure 4.6 but additionally compares a normalized version of the accumulated CPU times which tries to average out the measurement error: For each configuration for which the mean of all CPU time sample differences is less than the mean range of all measurement samples of the BroBase configuration, we do not use the measured time difference of the analyzer directly. Instead, we compute the average of ten samples at a time and use this average value as the analyzer's contribution for each sample within these 10 samples. For example, the mean contribution of the Finger analyzer is 0.004 sec. per sample. As this number is smaller than the mean range per sample (0.018 sec.), we compute the average measured contribution of the Finger analyzer for each ten samples (which is for an analyzer that does almost not contribute to the resource usage even closer to 0 than the individual samples). We then add this average to the measurements of the ten corresponding samples for the BroBase configuration to get the resource usage for processing the trace with the BroBase configuration plus the Finger analyzer. As result of this technique, we see in Figure 4.7 that there are no more samples where the accumulated CPU time sample is more than 0.2s higher than the measured sample. Using this heuristic, the mean relative error decreases to 3.5%, the median relative error to 2.8%.

Looking at the memory usage of the single analyzers, we use the same approach for checking the additivity of the single analyzers. For each instance of Bro running the

```
(ip[14:2]+ip[18:2]+tcp[0:2]+tcp[2:2]) -
   (((ip[14:2]+ip[18:2]+tcp[0:2]+tcp[2:2]) / 7) * 7) == 0x0
```

Figure 4.8: BPF filter expression for random connection sampling (factor 7)

BroBase configuration plus a single analyzer, we compute the difference in memory allocation between the instance with the additional analyzer switched on and the BroBase configuration. As expected, summing up these differences and the memory consumption of the instance running the BroBase configuration matches the memory usage of the Bro instance running the BroAll configuration well. The total memory allocation for the instance running the BroAll configuration is 461 MB. The sum of all memory usage contributions and the memory usage of the instance running the BroBase configuration is with 465 MB only 0.9% larger.

Here we have verified, that indeed the workload of Bro's individual application layer protocol analyzers is additive. Technically the imprecision of our measurements using the operating system's process accounting introduces a systematic error. We can reduce the influence of this error by averaging the measured CPU times over several seconds. In order not to suppress the effect of significant outliers, we do only average the measurements for analyzers that add a smaller workload than the operating system's accounting precision.

4.4.6 Random Connection Sampling

In a next step we verify our assumption that Bro's CPU workload scales with the number of connections processed. For this we capture a packet level full trace and apply random connection sampling to it with different sampling factors. Afterwards we run Bro instances on the different traces and compare the measured workloads.

First, we have to generate connection-sampled traces. Our connection sampling works as follows: We compute a checksum over source IP, destination IP, source port and destination port and determine the remainder for a division with a prime number P. Each connection in the network traffic now maps into one of the resulting P residue classes. Since parts of this checksum are fairly randomly distributed across connections (at least the source port number) and by division by a prime number the connections should be uniformly distributed over the residue classes. By now picking only the connections falling into one residue class we get roughly every P^{th} connection. We verified the approach by splitting Bro connection logs into residue classes with different primes and the connections were indeed equally distributed across the residue classes: Table 4.1 shows the distribution of 1.23 million connections into residue classes for different connection sampling factors. The approach is handy as it can be implemented without holding connection state. All state that is needed is contained in the IP, TCP and UDP header fields. Therefore we are able to implement this connection sampling approach into BPF filter expressions as shown in Figure 4.8. Since BPF expressions do not support the *mod* function, we compute the remainder R for the division of our checksum C with the prime

	sampling factor					
residue class	3	5	7	11	13	17
0	411,490	245,749	175,914	111,959	95,166	72,210
1	409,397	246,699	176,022	112,213	94,485	72,124
2	410,825	245,454	175,477	112,227	95,183	72,117
3		246,705	175,653	111,422	94,422	73,473
4		247,105	175,162	111,264	93,896	72,299
5			177,587	112,437	94,665	72,290
6			175,897	111,292	96,919	72,302
7				111,729	94,909	72,003
8				111,323	94,569	71,702
9				113,553	94,782	71,943
10				112,293	94,589	72,400
11					94,182	72,455
12					93,945	72,749
13						72,507
14						71,925
15						72,883
16						74,330

Table 4.1: Random connection sampling approach: 1.23 million connections split into residue classes with different sampling factors

number P to be $R = C - div(C, P) * P$, where *div* denotes a function for integer division (the function / in BPF syntax provides this functionality). The checksum C as described above is generated by the BPF expression `(ip[14:2]+ip[18:2]+tcp[0:2]+tcp[2:2])`. In the example, we use $P = 7$ and compare the remainder of the division R with `0`. The BPF expressions such as shown in Figure 4.8 can directly be processed by libpcap based programs like tcpdump or Bro.

In a first experiment we connection-sampled the trace we used before for comparing live Bro CPU usage against Bro reading data from the trace file for the BroBase configuration. In Figure 4.9 we see again the direct comparison between online and offline instances of Bro running the BroBase configuration like in Figure 4.4. Additionally the green '+' symbols denote the measurements taken for an offline Bro instance (also running the BroBase configuration) reading a connection-sampled (sampling factor 7) trace multiplied by the sampling factor. We notice that in general the extrapolated measurements are a tiny bit lower (the mean is 0.0034 seconds lower) but almost equal to the non connection-sampled trace. The most prominent difference is that the fluctuation for the Bro instance running on the connection-sampled trace is stronger. As the box plot in Figure 4.10 shows, this seems to be a trend: the larger the connection sample rate, the more fluctuation we get as indicated by the larger boxes. The reason for that is most likely that the measured CPU time values become very small if only few connections have to be analyzed: For example, in our unsampled trace Bro needs 0.15 sec. for one second

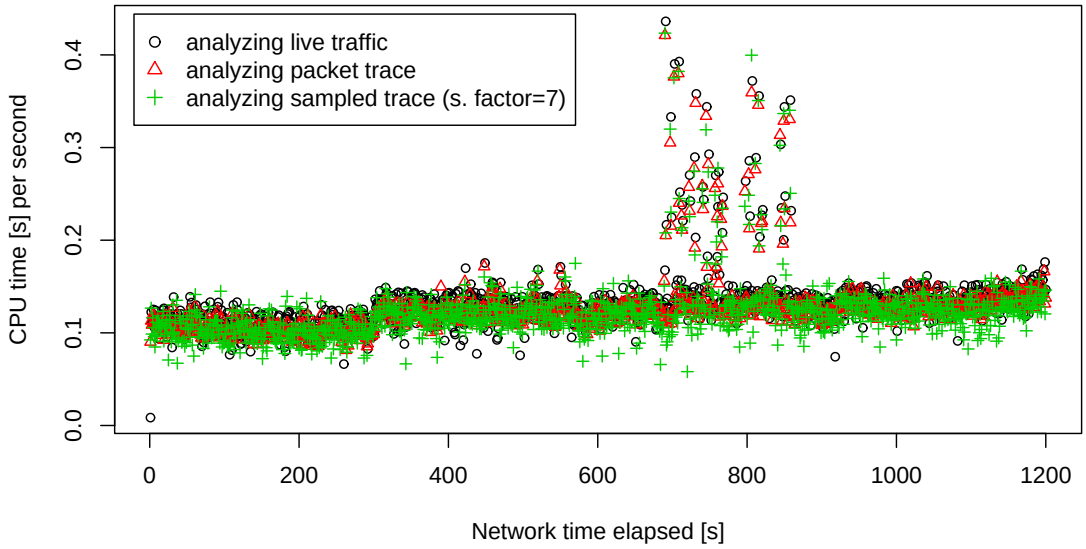


Figure 4.9: CPU usage for Bro instance running the BroBase configuration on connection-sampled packet trace

of network traffic analyzing all connections. With a sampling factor of 37 it will only need 0.15 *sec.*/37 = 0.004 *sec.* (since it has to analyze only one out of 37 connections). As discussed earlier, the operating system's process accounting is not precise enough for measuring these short time differences exactly, which leads to the extrapolated variation. However, if we look at the red line which is the median for the measurements of the online Bro instance, we see, that extrapolating the median by scaling the number of processed connections works well even for quite large connection sampling factors: The median of the extrapolated measurements is very close to the median of the measurements from the live Bro instance. On the other hand when we want to make statements about overload, we do not only need to know the mean/median CPU usage but also peak usage, and how often peaks in CPU usage occur.

For analyzing the influence of connection sampling further and on more complex configurations, we again use the 20 minutes excerpt of the `mwn-full-packets` trace. Here too, we ran the BroBase configuration with different connection sampling factors. For the connection sampling factor seven, Figure 4.11 shows that our stateless connection sampling approach works as expected with Bro: For all seven residue classes the distribution of consumed CPU time per second traffic is nearly the same.

The QQ-plot in Figure 4.12 compares the distribution of the measured CPU usage times for a Bro instance with the BroBase configuration running on an input trace containing all connections vs. the measured times of Bro instances running the same configuration on connection-sampled input trace files. For our 1200 measurement samples the plot shows 1200 quantiles of the CPU usage for the Bro instance running on an unsampled trace on the x-axis vs. the respective 1200 quantiles of Bro instances running

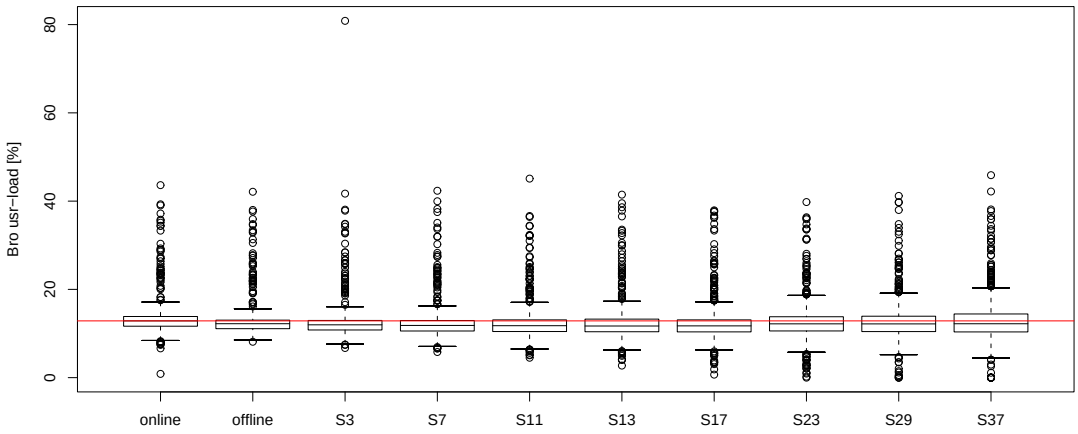


Figure 4.10: CPU usage for Bro instances running the BroBase configuration with different connection sampling factors

Configured sampling f.	1	7	11	17	31
min. memory usage [MB]	96.46	13.80	8.80	5.82	3.15
max. memory usage [MB]	96.74	14.88	9.05	6.16	3.49
real conn. sampling factor	1.00	6.99	11.01	16.99	30.96
memory allocation factor	1	6.5 - 7.0	10.7 - 11.0	15.7 - 16.6	27.6 - 30.7

Table 4.2: Variation in memory usage over 10 Bro instances running the BroBase configuration on the same packet trace

on random connection-sampled traces with sampling factors 7, 11, 17, and 31 on the y-axis. Figure 4.12 confirms, that overall Bro's workload running the basic configuration scales well with the number of connections. Even for large connection sampling factors like 31 there are only very few outliers. Up to the 99% quantile the directly measured CPU usage numbers match the ones extrapolated from the instances running on connection-sampled traces well. We note that our extrapolation slightly underestimates the real workload. For example, for the connection sampling factor of 7, the numbers are ca. 5% too low. Nevertheless we note that the extrapolated high quantiles (e.g., the 90% or the 95% quantile) are a good measure to decide whether the machine is overloaded too often.

Looking at the overall memory allocation of the different Bro instances does not yield very surprising results: Memory usage scales directly with the number of connections processed: Table 4.2 allows to compare the connection sampling factor with the Memory

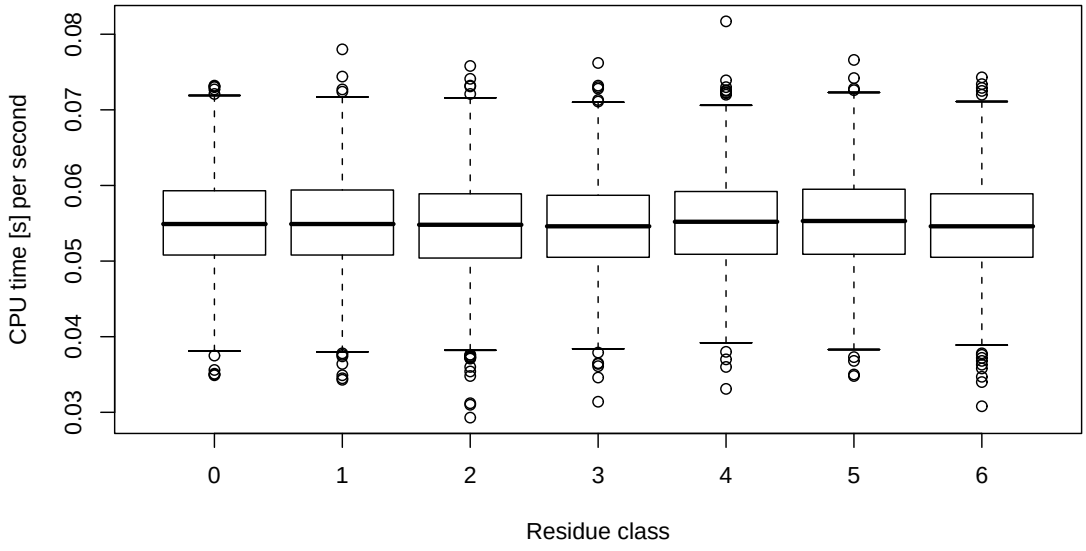


Figure 4.11: BroBase configuration connection sampling: distribution across residue classes for sampling factor seven

allocation factor of the corresponding Bro instance in detail. Remember, that our technique for random connection sampling relies on stateless filtering. Therefore the fourth row contains the "real" connection sampling factor computed as the number of connections in the unsampled trace divided by the number of connections in the sampled trace. One thing to note in Table 4.2 is the variation in memory usage. Running multiple Bro instances with the same configuration on exactly the same traffic (packet trace) does not result in the instances using exactly the same amount of memory. For our 10 instances running the BroBase configuration, the difference between the minimum total memory usage and the maximum total memory usage is ca. 300 KB for all sampling factors except seven[2]. In contrast to the variation in CPU usage, this variation does not scale with the total amount of memory allocated. So the absolute variation of the Bro instances running on the unsampled trace is the same as on all sampled traces. This means, that the relative variation grows significantly with the sampling factor, since overall memory usage linearly decreases with the number of connections. One origin of the variation in memory usage for multiple runs could be Bro's perfect hashing tables, that are randomly seeded. However running Bro with an command line option that fixes the seed did not eliminate the variation. We speculate that most of the variation is caused by different padding and different states of the memory management.

The next step is to check whether the load imposed by the analyzers scales linearly with the number of connections. We run the BroAll configuration (containing all 13

[2]Manual inspection of the measurements showed, that in only one of the ten measurements we see a significantly higher memory usage than in the other runs. Running another ten instances with the same configuration on the same trace did not reproduce the outlier.

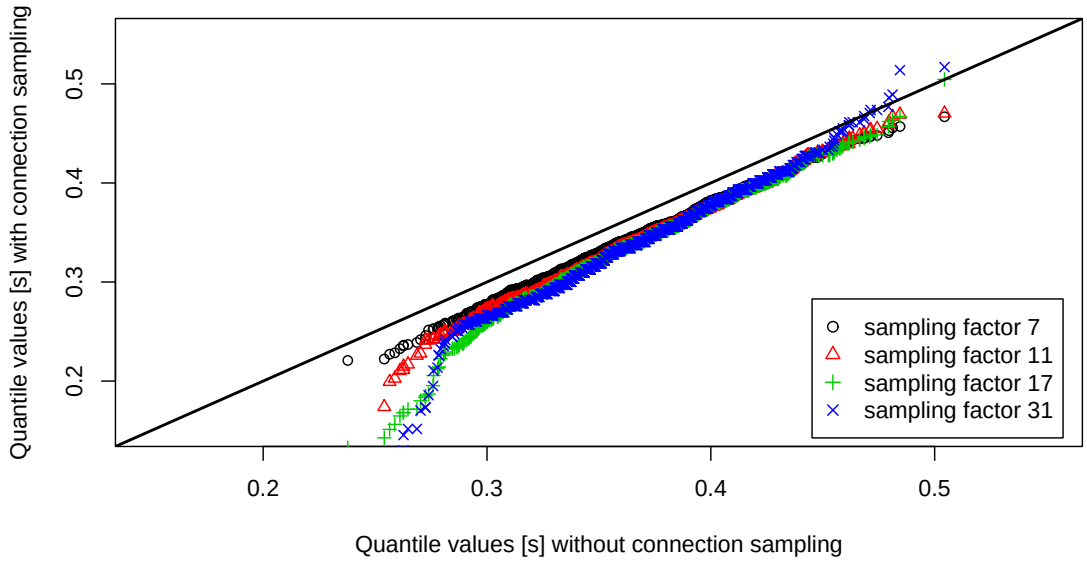


Figure 4.12: QQ plot BroBase configuration without sampling vs. different sampling factors

additional analyzers at once) on different connection-sampled traces. First the box plot in Figure 4.13 shows the distributions of the needed CPU time per second traffic for all residue classes when applying a connection sampling factor of seven. We note, that while the bulk of the distributions are the same, the outliers are significantly different. This indicates that while most connections are treated equally (meaning their processing needs approximately the same amount of CPU cycles), single connections need significantly more cycles. For example the strong outliers for residue class 0 all stem from a single connection on the POP3 port, causing processing spikes in the POP3 analyzer. Manual inspection shows, that this POP3 connection transfers a larger than usual volume (ca. 24 MB) within a relatively short timespan (ca. 9s). The implementation of Bro's POP3 analyzer seems not to be optimized to parse large connections very efficiently[3].

The QQ-plot in Figure 4.14 enables us to compare the analyzer workload of Bro instances running the BroAll configuration with different connection sampling factors (we use the residue class 0 for each sampling factor). As already indicated in Figure 4.13 the surplus work of one or more of our added analyzers does not always scale linearly with the number of connections processed: For instance for the sampling factor seven, the single POP3 connection that causes the processing spike happens to be in the sampled trace and the extraordinary high workload caused by this single connection is multiplied with the sampling factor. This results in a significant over-estimation of the extrapolated workload for the samples in which that connection is processed.

In general we note, that as for the basic connection handling, we underestimate the additional workload done by the analyzers. On the other hand the high quantiles tend

[3]This issue has been addressed in later Bro versions.

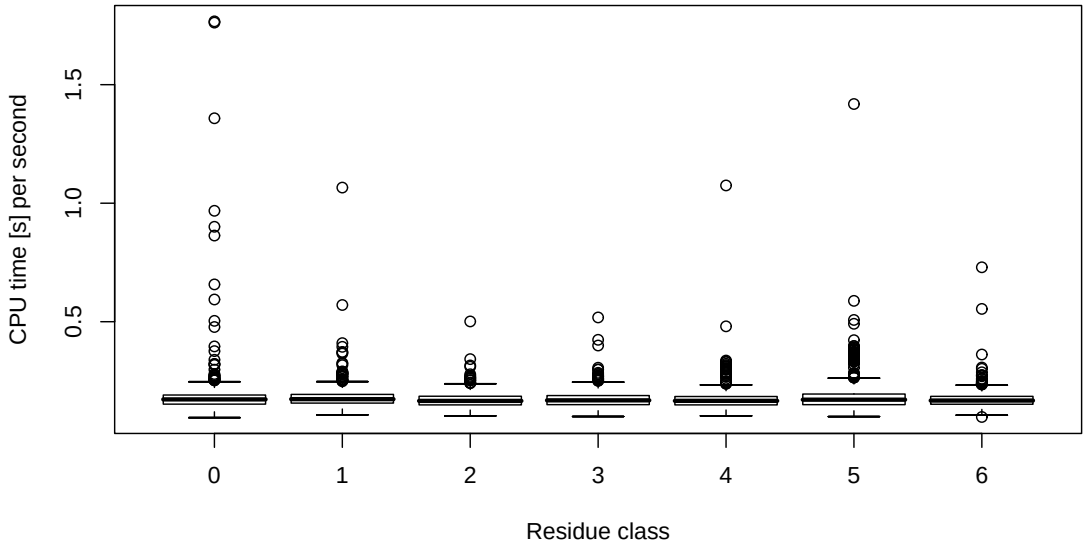


Figure 4.13: BroAll configuration connection sampling: distribution across residue classes for sampling factor seven

Configured sampling f.	1	7	11	17	31
max. memory usage [MB]	365	100	75	58	39
real conn. sampling factor	1.00	6.99	11.01	16.99	30.96
memory consumption factor	1	3.64	4.87	6.30	9.34

Table 4.3: Scaling of memory consumed by the cumulated analyzer state for the BroAll configuration

to be much less stable than for the basic connection handling. For all sampling factors, we note that the extrapolated samples with high CPU time are greater than in the unsampled case.

Remember from Section 4.3.2, that we do not expect the memory usage of all analyzers to scale directly with the number of random sampled connections. In case of the Bro default policy scripts, the reason is, that there are analyzers, which associate user level state not to every connection but to a set of connections. For example the HTTP analyzer allocates state for every new session. A session is defined to consist of all HTTP connections between two hosts within a short time interval. Our approach for stateless random connection sampling ignores such inter-connection semantics. In case of the HTTP analyzer, by random connection sampling, we decrease the mean number of connections per session. Put differently, this means, that the average amount of state associated per random sampled connection is larger than the average amount of state per connection in unsampled traffic.

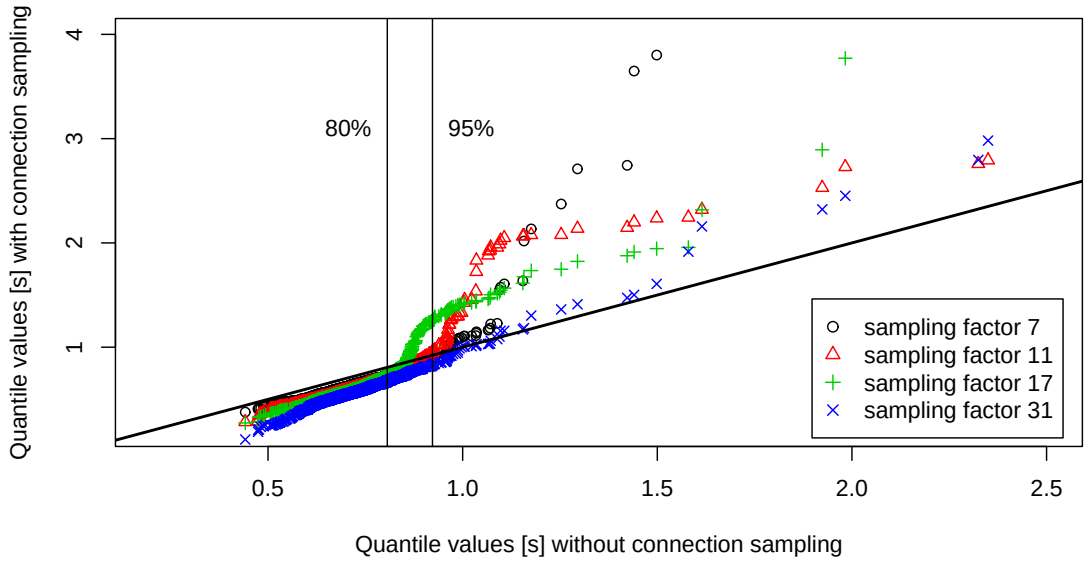


Figure 4.14: QQ plot measured analyzer workload (without sampling) vs. measured analyzer workload, different sampling factors

	Sampling Factor				
Analyzer	1	7	11	17	31
base	96.6	14.0	8.9	6.0	3.3
http-request	225.4	71.6	54.5	43.5	29.6
ssl	134.7	27.7	20.0	14.8	9.2

Table 4.4: Sum of memory allocation for analyzers that contribute significantly to memory usage and different sampling factors. Memory allocation numbers in MBytes.

For detecting when user level state does not scale directly with the number of connections, we can compare the total memory consumption for a configuration with different connection sampling factors: Table 4.3 shows such a comparison for the BroAll configuration. The row "memory consumption factor" shows, that memory does not scale linearly with the connection sampling factor. Note, that Table 4.3 does not show how the single analyzers contribute to the memory consumption pattern. Table 4.4 shows the memory usage for the BroBase configuration and the two analyzers included in the BroAll configuration that contribute significantly to the memory usage. In both, the HTTP and the SSL analyzer, script level state is responsible for the non-linearity in total memory allocation. Both associate state per session as discussed before.

For the statically coded core state, our code analysis showed, that it is associated to single connections and therefore scales per connection. For user-level state, we need to know, how the analyzer correlates it with the traffic (which in general can only be done by manually inspecting the policy script). Then we can (if there is a correlation to the

traffic) refine our model of Bro's resource consumption to extrapolate memory usage for the analyzer correctly. Overall memory usage of such an analyzer can be modeled to be a linear combination of the number of connections C and the number of entities E, to which the analyzer associates user state: $mem = x * C + y * E$. The coefficients x and y denote the average amount of memory allocated for each connection or entity, respectively and must be derived from the memory measurements of the Bro instrumentation (overall memory measurements are not sufficient any more). Another difficulty is to determine or assess the ratio of the number of E in the random connection-sampled trace and the full traffic. If the analyzer associates user state to sessions as defined above, we can determine the ratio of sessions in the random connection-sampled trace and in the full traffic by analyzing traces containing only connection control packets. We can model the session concept and count the sessions ($(IP_{source}, IP_{dest.}, Port_{dest.})$ tuples occuring within the session timeout) in the full traffic and the connection-sampled traffic.

Here we examined in detail in what circumstances Bro's workload, it imposes on the machine it is running on, scales linearly with the number of random sampled connections. We used random connection sampling to extrapolate Bro's resource usage from only a small subset of the traffic to the full traffic on a link. For memory usage, we verified, that it scales linearly with the number of connections for the BroBase configuration. For more complex configurations, several analyzers add state that is not associated to each connection. This results in the dependency between the number of random sampled connections and the overall memory usage to be non-linear.

4.4.7 Coupling Random Connection Sampling with Analyzer Combinations

Obviously the extrapolation of the analyzer workload from connection-sampled data introduces an error. Earlier we have seen, that summing up the workload of the individual analyzers also introduces an error relative to the measured CPU usage of the BroAll configuration. We now examine how large an error is produced by the combination of the two techniques. In Figure 4.15 we accumulate the additional workload per analyzer and compare the quantiles of the resulting sum with the quantiles of the directly measured values without connection sampling. The bright blue '$\diamond$' symbols in Figure 4.15 show again the direct comparison of the accumulated workloads vs. the measurements of the Bro instance running all analyzers at once as already shown in Figure 4.6. Overall we note that with larger connection sampling factors the number of extrapolated outliers significantly increases. For example, the 95% quantile (second vertical line) is overestimated with all sampling factors except 31.

As discussed earlier, (compare Figure 4.6 and Figure 4.7) for small measurements the relative error introduced due to the systems accounting imprecision may be large. We have also seen, that for a number of analyzers the additional workload even for the unsampled traffic is less than the variation between two instances running the same configuration. To reduce the impact of these measurement errors we again apply the normalization strategy described before: For all analyzers that in average add less workload than the mean range of all samples across 10 runs of the BroBase configuration, we consider averages over 10 samples as its contribution instead of the single samples.

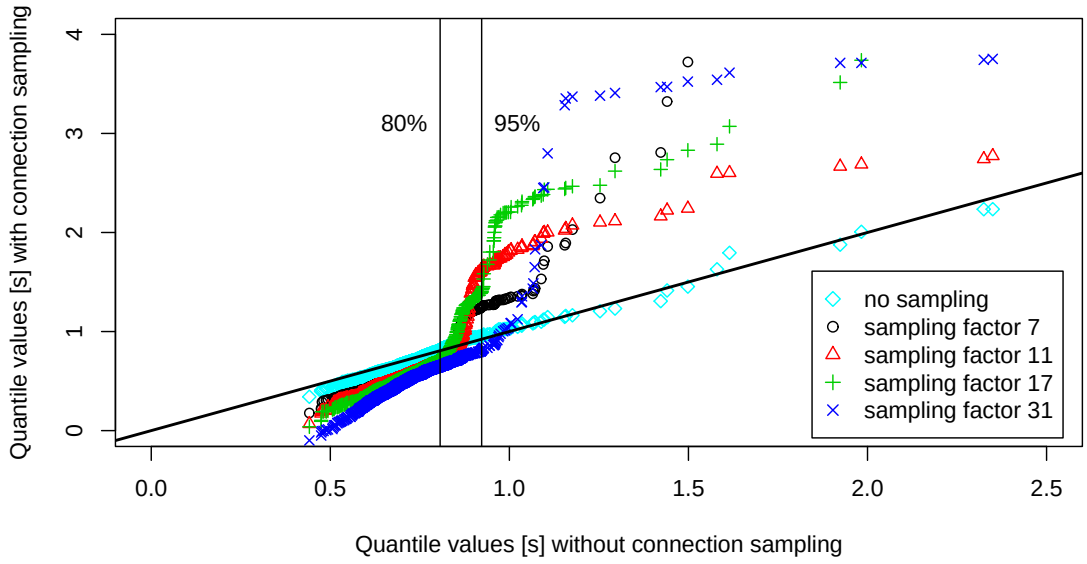


Figure 4.15: QQ plot measured analyzer workload (without sampling) vs. accumulated analyzer workload, different sampling factors

Figure 4.16 shows the result: For all sampling factors except for 17 the 95% quantile is not overestimated and for the higher quantiles the overestimation decreases significantly.

As expected, the error of our extrapolation is increased significantly by combining extrapolation from random connection-sampled traces and the approach to accumulate the workload of single analyzers. However, if we use our approach for normalizing the CPU measurements of the additional analyzers, the quality of the extrapolation increases substantially.

4.4.8 Summary

In this section, we performed systematic measurements of Bro's resource usage on live network traffic and on packet traces. First, we examined the interconnection between the CPU time accounted to the Bro process, the machine load and packet drops. We found, that it is infeasible to exactly predict when packet drops occur. On the other hand, our instrumentation of Bro is suitable to detect the lag in processing the packet stream, that often leads to packet drops. Using the same instrumentation we verified, that the CPU usage of a Bro instance running on a packet trace is not significantly different from one running on the corresponding live traffic. Based on this, we performed the following systematic measurements on 20 minutes packet traces. Since network traffic variation always produces single outliers in CPU usage, we focus on high quantiles (e.g., the 95% quantile) of CPU time measurement samples. We verified our assumption, that the additional resources that are needed to run individual Bro analyzers can be accumulated in order to compute the resource usage of a configuration that loads all these

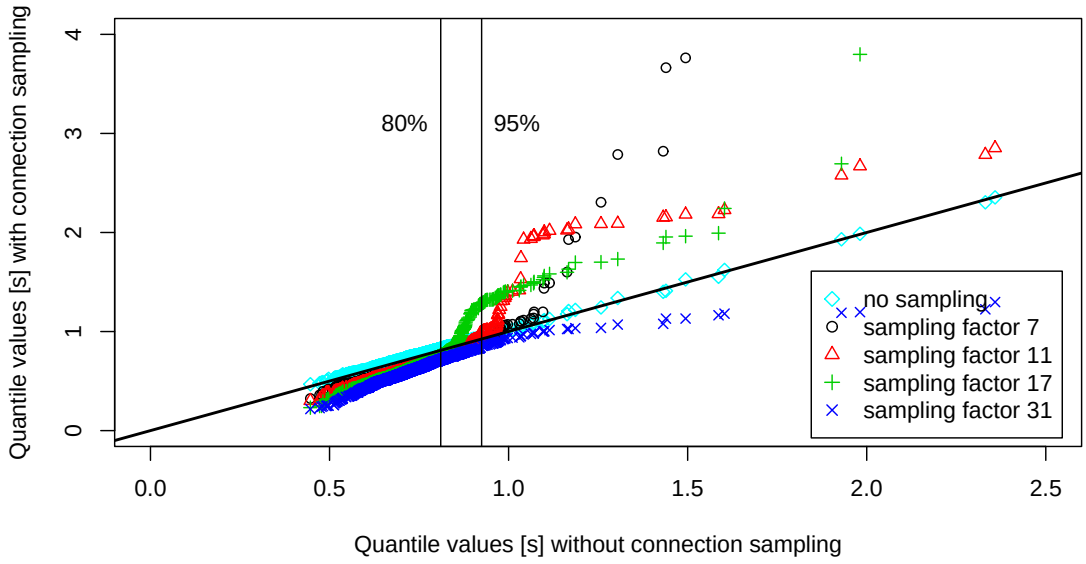


Figure 4.16: QQ plot measured analyzer workload (without sampling) vs. accumulated analyzer workload, different sampling factors, normalized

analyzers together. Then we examined how we can extrapolate Bro's resource usage by only measuring the resource usage of Bro instances that analyze a small subset of all traffic that crosses the link. In our approach we use random connection sampling, which works well for extrapolating the CPU time of simple and complex Bro configurations. Extrapolating the memory usage of complex configurations from random connection-sampled traffic is harder, as single analyzers associate state not to single connections but to sessions. For these analyzers, memory usage does not scale linearly with the number of random sampled connections. Finally we coupled our approach for extrapolating overall CPU time usage from random connection-sampled traces with our approach to accumulate the relative individual workload of a set of Bro analyzers to compute CPU time usage for complex Bro configurations. In Table 4.5 we summarize our systematic measurements and the problems with extrapolating Bro resource usage from random connection-sampled traces. Using our normalization technique which eliminates large measurement inaccuracies, we are able to yield good accuracy for extrapolating the 95% CPU usage quantile of complex Bro configurations.

4.5 A Toolsuite for Automatic Assessment of Bro's Resource Usage

In this Section, we present a toolsuite that actively supports an operator to determine an appropriate NIDS configuration for his network environment. The toolsuite consists of two auxiliary, NIDS independent, tools and the main resource usage measure and ex-

Analysis	CPU time	Memory usage	Problems with random connection sampling
connection handling	scales linear with #connections	scales linear with #connections	—
single analyzer	scales linear with #connections analyzed by that analyzer	core state: scales linear with analyzed connections by analyzer; user-state: scales with a linear combination of #connections and #other user-defined entities (e.g., sessions)	1. Contribution of analyzers for which the traffic mix contains few connections are more difficult to measure. 2. If the analysis of single connections is more expensive than average, extrapolation likely overestimates the actual cost 3. For analyzers that associate state per user-defined entity (e.g., session) overall memory usage does not scale linearly with the sampling factor.
complex configs	sum of connection handling and single analyzers' contributions	sum of connection handling and single analyzers' contributions	—

Table 4.5: Resource usage dependencies for Bro's components/configurations

trapolation tool. The two tools are used to automatically gather high-level information on the network environment in which the NIDS will be deployed: The first assesses the used link bandwidth and the second determines the network's IP address space. Both informations are used for the main tool: The idea behind the main tool is to automatically compare different NIDS configurations. It is based on systematic measurements of Bro's resource usage for analyzing packet traces. In order to reduce time and disk space required for the systematic measurements, we designed the tool to record and analyze connection-sampled full packet traces. The tool performs a sufficient number of measurements with different parameters so that the results can be combined in order to extrapolate the resource consumption when Bro runs with parameter combinations for which we do not perform measurements.

We start with discussing the two auxiliary tools. Then, for the main tool, we begin with discussing the trace capturing and analysis process on which our tool bases its measurements. Then we show how the tool composes the single measurements to extrapolate the resource usage of complex NIDS configurations. Finally we summarize the limitations of the tool.

4.5.1 Prepare Trace-Based Measurements

The first step towards a tool for automatically generating a configuration for NIDS is independent of any NIDS specifics. First, we assess the bandwidth that is currently used on the monitored link and second we present a methodology to determine the internal IP address space as a common parameter for NIDS monitoring the environment.

Determine Bandwidth of the Monitor-link

Our resource usage measurement tool is based on full packet traces. However, recording a full packet trace with commodity hardware is in high-volume network environments quite a challenge. To reduce the packet trace to a manageable size, we use random connection sampling. But first we need to have a rough idea of the traffic volume that crosses the link the NIDS is going to monitor in the end. For this we simply determine the bandwidth that is transferred via the link at the moment. We wrote a minimal libpcap application, that just counts and outputs the number of packets per second and the number of bytes per second on the link.

Detect Internal IP networks

Before we start our systematic resource usage measurements, it is possible to automatically determine an important parameter for today's NIDS: Network Intrusion Detection Systems usually have a notion of what IP address space to consider to be local. Here, we come up with a methodology to classify IP addresses observed on an Internet uplink into local addresses and remote addresses. We implemented this methodology into a tool which is provided with a packet trace of the environment. This tool is invoked from our main tool directly after capturing the connection-sampled full packet trace (see Section 4.5.2). However, we did not integrate its functionality into the main tool, as its functionality is also handy in other contexts.

The difference between local and remote hosts from the network operator's point of view is, that he has some control over the local machines but not over the remote ones. This information is vital, since for network operators the direction of traffic and the direction of attacks is very important: If one thinks of NIDS as "burglar alarms" it is tempting to only consider attacks directed towards the internal network in order to protect the internal hosts. The protection of the internal hosts is purely reactive: Since the operator usually has no control over the attacking system, he can only block the attack or ignore it if he is sure that this attack is not imposing any threat on the target system.

On the other hand, attacks originating from any host are a strong indication, that the originating host has been compromised. As soon as an internal host turns up to be launching attacks, the operator should check on that host and repair it. What at a first glance appears to be a courtesy to other Internet users is in fact important for the integrity of the internal network: compromised hosts are not trustworthy, may leak sensitive information and impose a severe threat to other hosts inside the network.

By doing a "post-mortem" analysis of how the host was compromised (if possible) the operator may protect other hosts from being compromised in the same way.

Although the control policy of the network environment specifies what hosts are internal and what hosts external, there are also technical indications. Given that NIDS are usually deployed at some access link to the Internet (see Figure 2.1) the internal hosts are located on one end of this point-to point link whereas the external ones (or the Internet) are connected via the other end. In order to configure NIDS with the local network one has to associate IP addresses with the local end of the link. On the link-layer there is also an addressing scheme independent of the IP addressing above. Let the internal link-endpoint have link-layer address A and the external link-layer address be B. Each IP address is either internal or external, e.g., if IP address IP_1 is internal, on the link we can observe packets going from A to B with IP_1 as source or packets from B to A with IP_1 as destination. This methodology can be applied to any number of point-to-point access links: Each access link has a local and a remote link-endpoint associated with a set of IP addresses. By simply collecting all IP addresses associated with the internal endpoints of the link, we can get all used internal IP address blocks by IP aggregation.

An important question is: how long to collect IP addresses. The problem is that if we run our algorithm for a too short time interval, we may miss single internal addresses, since they may be idle during the observation. Ideally an NIDS would use this methodology constantly to its traffic and dynamically update its notion of internal networks.

To get a snapshot of the IP address space of a network environment we use a heuristic that aggregates single IP addresses to larger IP address blocks more aggressively: Instead of waiting until all addresses of a contiguous address block have been observed, we aggregate two already observed IP address blocks if they are not too distinct. The notion of "too distinct" can be adjusted by the user, meaning that the user specifies a shortest allowed prefix length. For example, we observed the IP addresses 10.0.0.1, 10.0.0.234, 10.0.1.1, 10.0.1.110 and 10.0.2.100 on the internal endpoint of the monitored link. The maximum size of the aggregated IP address blocks given by the user is /24. With our heuristic we aggregate the observed IP addresses to 10.0.0.0/24, 10.0.1.0/25 and 10.0.2.100/32. Using this heuristic we only miss single addresses if a larger address block is inactive during the time our tool monitors the link. In the MWN the heuristic works well: We ran our tool on 5 minutes of connection control packets (captured at 2pm local time) with a shortest allowed prefix length of 16.

A comparison of its output with our network operator's list of internal networks shows, that the tool detected all seven internal /16 address blocks correctly. However, according to our operator's list, in the 141.39.0.0/16 address block only 141.39.128.0/18 and a 141.39.240.0/20 are in use. Our heuristic implemented in the tool aggregates these to 141.39.128.0/17. We see similar problems for the smaller address blocks that are assigned to the MWN and actively in use: In the worst case the rather aggressive heuristic to aggregate address blocks to /16 prefixes aggregates three individual /24 address blocks (153.96.48.0/24, 153.96.49.0/24 and 153.96.185.0/24) to their longest common prefix (153.96.0.0/16). In total eleven assigned and used /24 address blocks are wrongly aggregated to a /16, a /17 and a /21 address block. Of the remaining /24 and smaller address blocks routed into the MWN, the tool detected and aggregated 14 correctly. Five

/24 blocks were not detected as they did not show any activity during our five minutes trace.

This example shows, that the parameter for the address block aggregation heuristic needs to be chosen carefully. Although the rather aggressive setting of /16 generates a nice compact list of internal networks, it includes rather large address blocks in the "internal networks" set that definitely are not internal. As depicted in the example, this happens only if the network is assigned small scattered address blocks within one larger block. To counter this, one could either choose the parameter of the tool more conservatively (e.g. /24) or the tool's simple heuristic could be refined. For instance, one could implement a step-wise approach: The refined heuristic would need a parameter for the minimum assigned address block size and an additional parameter for the maximum size of assigned address block. The tool then would only aggregate to address blocks between the minimum and maximum sizes if the minimum blocks that were detected cover a significant fraction of the larger IP address block. Thus the scenario from our example, where three scattered /24 networks aggregated to a /16 address block could be avoided.

So far, we assumed that the user knows the link-layer address that belongs to the internal end of the access link. However we implemented another heuristic, that automatically determines which link-layer address is to be considered the internal one: For any network environment we assume, that the Internet, that is the network connected through the external end of the access link, consists of a broader set of IP address blocks than the internal network. Normally this causes access to many different IP address blocks from internal users and/or from scattered external IPs to single internal services. Therefore, we consider that link-layer address to be the internal one which has the smaller number of aggregated IP address blocks associated.

4.5.2 Main Tool: Capture Trace as Measurement Base

The resource usage measurements of our main tool are based on full packet traces. Consequently, its first task is to capture a random connection-sampled full packet trace. This will be used to perform our systematic measurements of Bro's resource usage on the given machine. To get an idea of the traffic mix and the short-term variability of the network traffic, the trace needs to cover a sufficient time interval. Additionally, the time the trace covers needs to be long enough for the NIDS to get past its startup phase in order to get meaningful results for the NIDS resource usage measurements. While in general, the length of the trace is a parameter in our tool, for our experiments here we choose to capture traces covering 20 minutes of traffic. Our experiments in Section 4.4 show, that a 20 minutes connection-sampled trace captures traffic mix and short-time variability of the network environment in appropriate detail and is suitable to measure the NIDS' resource usage after its startup phase.

In the following we refer to this trace as the *main analysis trace*: it will be the one we base all our measurements with the Bro NIDS on. Therefore it should ideally contain all traffic that occurs on the network within our 20 minutes capture interval. However this is not even feasible in gigabit networks that are used to only half their capacity: A network

transferring some 400 MBit per second results in a 20 minute trace of approximately 60 GB of data. First we cannot assume that the machine we are running on has this much disk capacity and second it is very difficult with standard PC hardware to raw-capture these volumes without suffering packet losses. Therefore we use our connection sampling approach as outlined in Section 4.4. We aim at capturing a user provided trace volume by determining the sampling factor with $S = B * D/V$ where S is the sampling factor, B the transferred bytes per second as determined by our auxiliary tool described in Section 4.5.1, D the duration of our trace and V the user specified volume. Note that with this approach, we are not able to capture a trace of exactly the size the user specifies since first the network bandwidth always fluctuates and we are using only a ten second peak and second we cannot use any connection sampling factor: in fact we use the next greater prime number. Both inaccuracies lead to the trace being rather smaller than the specified limit.

Since our random connection sampling is completely stateless, an adversary or bad luck could lead to a false estimation of the real number of connections from our sampled connection trace. An adversary could, if he knew the sampling factor and the residue class we look at, generate a large number of connection attempts that all fall into this residue class, therefore biasing our extrapolation. To avoid this class of problems we capture two additional packet traces in parallel to our main analysis trace:

1. Connection control trace: all SYN, FIN and RST packets

2. Packet-sampled trace: roughly one-per 4096 packets

These two traces are used to determine the real connection sampling factor: We run Bro instances configured to do connection analysis and output one-line connection summaries for the main analysis trace and for the trace containing all connection control packets. By dividing the number of connections analyzed in the main analysis trace by the number of connections in the connection control trace we get a more accurate number for the connection sampling factor if for some reason our stateless approach for connection sampling is inaccurate. Using the connection control trace with Bro, a connection turns only up in the connection summaries, if either a SYN, a FIN or a RST is observed for the connection during the recording. For long connections this may not be the case: connections that start before the recording interval and end after it. Nevertheless these may contribute packets (and therefore also connections) to the main analysis trace. Therefore we correct the number of connections analyzed in the connection control trace by adding the number of combinations of IP addresses and TCP ports, that do not appear in the connection summaries but in the packet-sampled trace. Note, that our approach for packet sampling is similar to the one for random connection sampling. We use a BPF filter expression, that only accepts packets for which the condition $mod(IP\_checksum, 4096) == 0x0$ evaluates to true. In some cases this may have different properties than periodic packet sampling.

So far we did never observe a major discrepancy between the chosen sampling factor and the real sampling factor. However, for our extrapolation we use the real sampling

Input:

`dur`	Duration [sec] of main analysis trace (User-specified)
`volume`	Limit [bytes] of disk space to use for main analysis trace (User-specified)
`bandw`	Peak bandwidth [bytes/sec] of the link

```
sFactor = next_prime_number_greater_than((dur * bandw)/volume)

start_packet_record_process(filter=conn_sample(sFactor), output=main_trace)
start_packet_record_process(filter=conn_control_pkts, output=connCtl_trace)
start_packet_record_process(filter=pkt_sample(4096),  output=pktSample_trace)
wait(dur)
terminate_all_record_processes()

sampledConnections = count_number_of_conns(in=main_trace)
allConnections = count_number_of_conns(in=connCtl_trace)
longConnections =
    count_number_of_conns(in=(pktSample_trace without connCtl_trace))

realSampleFactor = (allConnections + longConnections) / sampledConnections
```

Output: `main_trace`, `realSampleFactor`

Figure 4.17: Pseudo-code capturing traces and determine real connection sampling factor

factor, computed as described here. Figure 4.17 summarizes our methodology for capturing the trace set and computing the real sampling factor as pseudo-code.

4.5.3 Main Tool: Run Measurements and Extrapolate Resource Usage

The goal of our main tool is to determine what parameter combination does not overload the system if analyzing a traffic mix and volume as captured in our main analysis trace. Thus we aim at finding the maximum set of analyzers that does not impose overload onto the machine. Additionally, we aim at determining values for the connection timeouts that do not drive Bro to use more memory than the user specifies.

Having collected a trace with the traffic mix from the environment our NIDS will be deployed in, we can now exploit the results of our code analysis (see Section 4.3.1). We present a methodology for systematically performing measurements on the basis of the main analysis trace. We implemented this methodology into our main tool which interacts with the user and automatically generates a Bro parameter set which is suitable to analyze the traffic sample with the provided resources.

As baseline measurement we run a Bro instance with the BroBase configuration which performs only TCP analysis. Connection timeouts are configured rather conservative: Connection attempts are expired after 300 sec., the inactivity timeout is disabled, meaning that once established connections are never expired if no termination is seen. We add our instrumentation in order to measure CPU and memory consumption every

trace second.

The additional analyzers that are potentially interesting for the user are supplied in a list of policy scripts to the tool (see Section 4.5.4). These policy scripts are supposed to contain all analyzer specific configuration. At the moment, the tool does not aim at configuring parameters of the individual analyzers, since this would imply to automatically generate parametrized performance models for the single analyzers. For each analyzer in the list we run a Bro instance with the BroBase configuration plus the additional policy script (compare Section 4.4).

For all instances, we multiply the measurement samples with the connection sampling factor. For CPU time, we only take the measurements after 300 seconds into account: At that time the attempt timers start to expire and cause additional workload (see Section 4.4). For all instances but the baseline measurement we look only at the additional CPU time and additional memory consumption as compared to the baseline. The pseudo-code in Figure 4.18 outlines our approach for the systematic measurements of Bro resource usage on the main analysis trace.

Adapt Configuration to CPU Load

For CPU time we aim at a configuration (that is, a set of policy scripts) for Bro that does the most sophisticated analysis while not losing any packets. Thus we want to include as many analyzers into the configuration as seems possible without overloading the system. Having the contribution in terms of CPU time for each analyzer at hand, we can extrapolate the CPU time usage for a maximal configuration with all analyzers loaded (compare Section 4.4). Furthermore if our measurement shows, that a single analyzer already overloads the system we can exclude that analyzer from the set of possible analyzers. If the maximal configuration (without the already excluded analyzers) overloads the system, our program asks the user which analyzer to omit. It helps the user's decision by stating the extrapolated CPU load for the maximal configuration and the contributions of the single analyzers. In Figure 4.19 we summarize our approach for adapting a Bro configuration to CPU load in pseudo-code.

One of the central questions is: how can we decide that the system is overloaded? For this we summarize our per second measurement by looking at a user specified quantile of CPU usage per sample. The idea behind this is, that the system has a threshold CPU load, e.g., 90% (corresponding to 0.9s CPU time per 1.0 second trace time sample) at which we fear that a "lag" in processing the packets in real time is building. If for too many samples the extrapolated CPU usage is over the threshold, the probability of packet losses increases. The user defined quantile specifies, for what percentage of samples the CPU time of the Bro configuration under test has to stay below that threshold. Thus a high quantile, e.g., the 95% quantile still allows single surges in CPU time usage but for most of the time the load of the Bro process has to stay below the threshold.

Input:

`main_trace`	Random connection-sampled main analysis trace (see Figure 4.17)
`realSampleFactor`	Real connection sampling factor of `main_trace` (see Figure 4.17)
`base_config`	Bro baseline configuration (User-specified)
`config_list`	List of Bro configurations (analyzers) (User-specified)

```
bro_filter = determine_Bro_packetFilter(BroConfig=base_config)
filter_trace = prefilter_trace(input=main_trace, filter=bro_filter)
base_res_prof = run_Bro_profile_resource(BroConfig=base_config,
                traffic=filter_trace) * realSampleFactor

#needed for detecting error in extrapolating script level state:
filter_trace_sub3 = prefilter_trace(input=filter_trace, filter=conn_sample(3))
base_res_prof_sub3 = run_Bro_profile_resource(BroConfig=base_config,
                       traffic=filter_trace_sub3) * realSampleFactor * 3

foreach config in config_list
{
   bro_filter=determine_Bro_packetFilter(BroConfig=config)
   filter_trace = prefilter_trace(input=main_trace, filter=bro_filter)

   res_profile = run_Bro_profile_resource(BroConfig=config,
                   traffic=filter_trace) * realSampleFactor
   rel_res_prof_tab[config] = res_profile - base_res_profile

   #needed for detecting error in extrapolating script level state:
   filter_trace_sub3 = prefilter_trace(input=filter_trace,
                         filter=conn_sample(3))
   res_prof_sub3 = run_Bro_profile_resource(BroConfig=config,
                     traffic=filter_trace_sub3) * realSampleFactor * 3
   rel_res_prof_sub3_tab[config] = res_prof_sub3 - base_res_prof_sub3
}
```

Output:

`base_res_prof`, `rel_res_prof_tab`
`base_res_prof_sub3`, `rel_res_prof_sub3_tab` (see Figure 4.21)

Figure 4.18: Pseudo-code for systematic measurements of Bro resource usage

Input:

`base_res_prof`	Resource usage profile of Bro baseline configuration (see Figure 4.18)
`config_list`	List of Bro configurations (analyzers) (see Figure 4.18)
`rel_res_prof_tab`	Table of relative resource profiles (one for each configuration in `config_list`) (see Figure 4.18)
`cpu_limit`	Limit of CPU time [sec.] to spend for one second of traffic (User-specified)
`limit_quantile`	Quantile of CPU samples that need to adhere to `cpu_limit` (User-specified)

```
if (exceeds_CPU_Limit(prof=base_res_prof)){abort}

foreach config in config_list
{
   if (exceeds_CPU_Limit(prof=(base_res_prof + rel_res_prof_tab[config])))
   {
      remove_element(config_list[config])
      remove_element(rel_res_prof_tab[config])
   }
}

while (exceeds_CPU_Limit(prof=sum(base_res_prof, rel_res_prof_tab)))
{
   show_list_of_CPU-time_contributors(config_list)
   config_to_remove = ask_user_which_to_remove()
   remove_element(config_list[config_to_remove])
   remove_element(rel_res_prof_tab[config_to_remove])
}

function exceeds_CPU_Limit(prof)
{
  return (quantile(prof, limit_quantile) > cpu_limit)
}
```

Output: `config_list`, `rel_res_prof_tab` (modified)

Figure 4.19: Pseudo-code for adapting Bro configurations to CPU load

Adapt Configuration to Memory Consumption

For memory consumption our first goal is to adapt the connection state timeouts inside the Bro core to values that can cope with the arrival rate of connections in the given environment. Given our findings in Section 4.3.1 and Section 4.4.6, running the BroBase configuration, there are three different types of connections that are treated differently: *(i)* unsuccessful connections or connection attempts, *(ii)* normally established and normally terminated connections, and *(iii)* normally established but not terminated connection. The first type is handled by the connection compressor and the *attempt* timer. For the second the state handling is done implicitly, state is safely expired soon after the connection is terminated. The third type of connections are expired by the inactivity timer (see Section 3.7.1).

Given our coarse instrumentation for determining the overall memory allocation we cannot directly derive the memory consumption for the three different types. However, Figure 4.20 illustrates, that Bro's memory allocation can be divided into three phases given a typical trace, where each of the three connection types has an almost constant arrival rate. In phase one (red line) all three types of connections need to allocate memory for their state. Memory consumption grows rapidly. However this phase does not last long: It ends as soon as the number of freshly arriving normal connections is the same as the number of terminated and expired normal connections. This is (given an ordinary connection-length distribution) usually the case after ca. 30-60 seconds. In phase two (green line) only connection attempts and non terminated connections need to allocate new memory for their state. The connection compressor collects all unsuccessful connection attempts until they start to be expired after 300 sec. At that time the connection compressor does not need to allocate more memory, given that the arrival rate of unsuccessful connections remains roughly the same and phase three begins. In this phase (blue line) the only connection type that allocates fresh memory are the connections that do not see proper termination.

From these three phases we can derive a first approximation for the attempt timer and the inactivity timer: By assuming that connection arrival rate for all three types of connections remains roughly the same, we know from the slope in phase three, how much memory per second is needed for connections that are not terminated. Provided with a user specified memory budget for these connections we can compute the maximum time we are allowed to store each of those connections without expiring it: the length of the inactivity timeout. By subtracting the mean memory allocation in phase three from the corresponding value in phase two, we get the memory allocation per second for the unsuccessful connections handled by the connection compressor. Again we need to be provided with a memory budget for this type of connections by the user and can so derive a value for the attempt timeout.

For assessing the additional memory consumption of the single analyzers, we have the problem, that the internal Bro instrumentation does not account the additional per connection state kept in the event engine. This makes it difficult to distinguish between "core state" and "user state". For the time being, we use Bro's instrumentation for approximately measuring user state memory consumption as described in Chapter 3. It

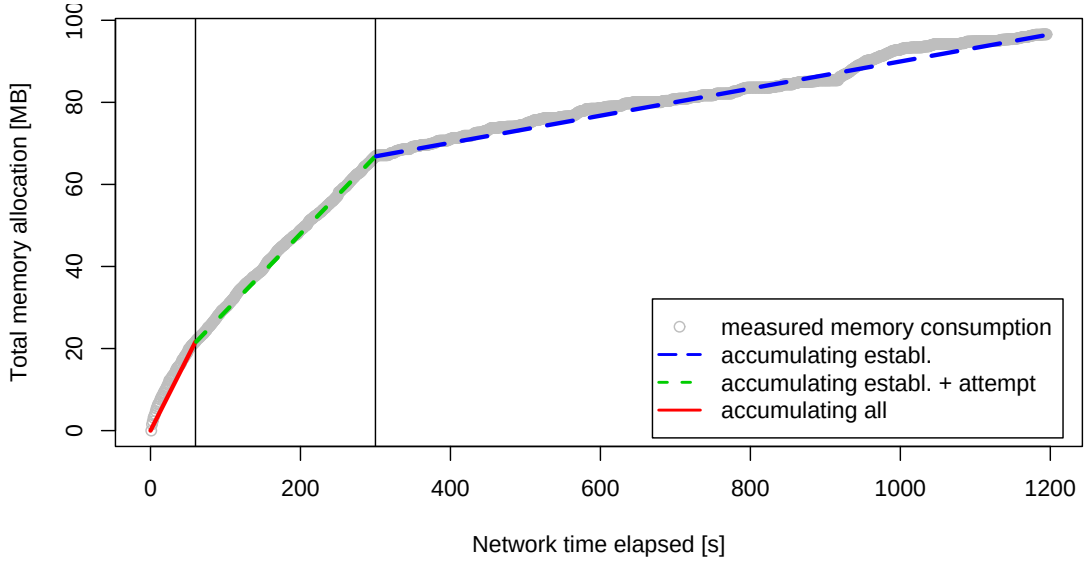


Figure 4.20: Typical memory footprint for first 1200 seconds running the BroBase configuration (Attempt-timeout: 300 sec., Inactivity-timeout: disabled)

provides us with a rough number of bytes occupied by user-level data structures that are larger than 100 KB, which we subtract from the total memory consumption of the Bro instance. The difference between the result and the total memory consumption of the instance running the BroBase configuration is considered the contribution of the analyzer to the "core-state". This number is used to adapt the connection timeouts to the steeper slopes (compared to Figure 4.20) in memory consumption. Note that this approach is not accurate due to imprecise numbers for the memory consumption on the script-layer (compare 3.4) and variability in memory consumption across multiple runs (compare 4.4.6).

Extrapolating user state memory consumption for the additional analyzers is even more difficult. One problem comes from our approach to use random connection-sampled traces: As we have seen in Section 4.4.6, the additional memory of the analyzers does not necessarily scale linear with the number of random sampled connections. Therefore, we "subsample" the connection-sampled main trace with sampling factor three. In fact, we use the same random connection sampling approach as before again (see Figure 4.18). For each analyzer, we determine the total memory consumption of a Bro instance running on the subsampled trace. Now, we multiply this number with the subsampling factor three. If this yields approximately the number for the memory consumption of a Bro instance running the same configuration on the main trace, the analyzer's memory consumption scales linearly with the number of randomly sampled connections. Then we can extrapolate memory usage by multiplication with the connection sampling factor as with the CPU usage. Otherwise, the tool outputs a warning, that memory usage does not scale directly linear with random sampled connections.

Input: (unless noted otherwise, see Figure 4.18)

`base_config`	Bro baseline configuration
`base_res_prof`	Resource usage profile of Bro baseline configuration
`config_list`	List of Bro configurations (output from Figure 4.19)
`rel_res_prof_tab`	Table of relative resource profiles (one per configin `config_list`)
`base_res_prof_sub3`	Resource usage profile of Bro baseline config. on subsampled trace
`rel_res_prof_sub3_tab`	Table of relative resource profiles on subsampled trace (one per config. included in `bro_cpu_config`)
`measure_attempt_timer`	Value of Attempt-timeout during systematic measurements
`coremem_limit`	Memory limit for core state (User-specified)
`coremem_fract_ATTEMPT`	Fraction of core-memory for attempts (User-specified)
`scriptmem_limit`	Memory limit for user state (User-specified)

```
if (core_mem_usage(in=base_res_prof, at=30s) > coremem_limit or
    script_mem_usage(in=base_res_prof, at=30s) > scriptmem_limit) {abort}

foreach config in config_list
{
   mem_prof = base_res_prof + rel_res_prof_tab[config]
   if (core_mem_usage(in=mem_prof, at=30s)>coremem_limit or
       script_mem_usage(in=mem_prof, at=30s)>scriptmem_limit)
   {
     remove_element(config_list[config])
     remove_element(rel_res_prof_tab[config])
   }
   elsif (is_not_approx_equal(overall_memUsage(in=rel_res_prof_tab[config])),
                              overall_memUsage(in=rel_res_prof_sub3[config])))
      { warn("script level state of config does not scale linearly") }
}
mem_prof = sum(base_res_prof, rel_res_prof_tab)
coremem_limit = coremem_limit - core_mem_usage(in=mem_prof, at=30sec)
coremem_limit_attempt = coremem_fract_ATTEMPT * coremem_limit
coremem_limit_establ = coremem_limit - coremem_limit_attempt

slope_establ = mem_alloc_slope(start=measure_attempt_timer, end=end_of_measure,
                               prof=mem_prof))
slope_attempt = mem_alloc_slope(start=30, end=measure_attempt_timer,
                                prof=mem_prof)) - slope_establ
inact_timeout = coremem_limit_establ / slope_establ
attempt_timeout = coremem_limit_attempt / slope_attempt
bro_config = generate_Bro_config(base_config, config_list,
                                 inact_timeout, attempt_timeout)

function mem_alloc_slope(start, end, prof)
{
   alloc = core_mem_usage(in=prof, at=end) - core_mem_usage(in=prof, at=start)
   return (alloc / (end-start))
}
```

Output: `bro_config`

Figure 4.21: Pseudo-code for adapting Bro connection timeouts to memory limits

The tool does not suggest expire timeouts for user state. One reason is the difficulty in extrapolating memory usage described before. Another problem is, that dependencies between user defined data structures had to be automatically analyzed which is an extremely complex task. The tool does however reject single policy scripts that defined user state data structures that grow larger than the user specified script-memory limit within 30 seconds. The pseudo-code in Figure 4.21 gives an overview on our methodology to analyze Bro memory usage and adapt the connection timeouts accordingly.

4.5.4 Example Run

We started our tool for a test run on May 23 2006 at 4 pm. We configured the capture parameters to 20 minutes packet traces with a disk space budget of 5 GB for the main analysis trace. The peak bandwidth used on the link during a 10 seconds snapshot was determined to be 695 MBps. A 20 minutes full packet trace captured on a network using that bandwidth would result in approximately 100 GB of data. Consequently our tool uses a connection sampling factor of 23 (next larger prime to the needed connection sampling factor of ca. 21). The connection-sampled trace, that the tool subsequently captured needs exactly 5 GB disk space. The two additional traces that are captured in parallel together occupy another 847 MB: 752 MB for the trace with all connection control packets and 95 for the packet-sampled trace (sampling factor: 1 packet in 1024). The real connection sampling factor is 23.047.

The systematic measurements are configured by a file listing all Bro configurations that have to be measured individually. Each of these configurations is represented by one line in the file as shown in Figure 4.22. The `broscripts` are the policy scripts that have to be loaded for the configuration. Bro command-line parameters that are necessary for this configuration can be supplied via the second field. The last two fields are identification numbers for the configuration. The `GID` field denotes different configuration spaces. The tool considers all configurations with the same GID to be independently combinable. The ID is a number that allows the tool to unambiguously identify a configuration. The configuration with ID 0 in each group is used as the baseline configuration.

For our test run we configured one group (GID 0) and 17 individual configurations. The baseline configuration (ID 0) was defined to be the BroBase configuration. Each of the 16 other configurations consisted of the BroBase configuration plus one single analyzer.

After the systematic measurements were run, our tool started to interpret the output of our instrumentation. Figure 4.23 shows the output of the tool for this last phase. First the tool outputs the active user specified limits. Then, each Bro configuration-line is checked separately. In our example for the Bro configurations with IDs 1, 9, 10 and 15 (the scan, http-request, http-reply, and ssl policy scripts) the tool detected

```
<broscripts>; <bro command-line parameters>; <GID>; <ID>;
```

Figure 4.22: Configuration line for the tool to extrapolate resource usage of complex Bro configurations

```
Active Limits:
CPU: 90.00% of samples < 80% CPU load
MEM: Connection state: 500 MB, Script-level state: 300 MB
SCALING: GID 0 ID  1 Script-level state does not scale directly with #conn.
SCALING: GID 0 ID  9 Script-level state does not scale directly with #conn.
SCALING: GID 0 ID 10 Script-level state does not scale directly with #conn.
LIMIT:   GID 0, ID 10: Resource usage too high for limits
         (90% CPU-quantile: 247.75, connMem: OK,  scriptMem: OK)
SCALING: GID 0 ID 15 Script-level state does not scale directly with #conn.

now configuring Group 0
current config (IDs 0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16): CPU usage too high.
90% CPU-load quantile: 112%
Choose analyzer to deactivate:
ID   avg. CPU-load              Scripts
     contribution
-------------------------------------------------------------------------------
9    29%   broconf-timers broconf-profile weird tcp noscan intern http-request
15    8%   broconf-timers broconf-profile weird tcp noscan intern ssl
13    7%   broconf-timers broconf-profile weird tcp noscan intern pop3
11    6%   broconf-timers broconf-profile weird tcp noscan intern smtp
12    5%   broconf-timers broconf-profile weird tcp noscan intern ssh
1     2%   broconf-timers broconf-profile weird tcp intern
4     1%   broconf-timers broconf-profile weird tcp noscan intern ftp
2     0%   broconf-timers broconf-profile weird tcp noscan intern login
5     0%   broconf-timers broconf-profile weird tcp noscan intern finger
14    0%   broconf-timers broconf-profile weird tcp noscan intern irc
8     0%   broconf-timers broconf-profile weird tcp noscan intern tftp
16    0%   broconf-timers broconf-profile weird tcp noscan intern gnutella
3     0%   broconf-timers broconf-profile weird tcp noscan intern ident
7     0%   broconf-timers broconf-profile weird tcp noscan intern frag
6     0%   broconf-timers broconf-profile weird tcp noscan intern portmapper

your choice: 9
current config (0 1 2 3 4 5 6 7 8 11 12 13 14 15 16) adheres to CPU limit.

configuring connection timeouts for GID 0
ConnMem ID 0: after 30s: 19.302MB slope30-300: 0.193MB/s slope300+ 0.037MB/s
ConnMem ID 1: after 30s: 2.845MB slope30-300: 0.004MB/s slope300+ 0.005MB/s
ConnMem ID 2: after 30s: 0.000MB slope30-300: 0.033MB/s slope300+ -0.000MB/s
ConnMem ID 3: after 30s: 1.314MB slope30-300: -0.000MB/s slope300+ -0.000MB/s
ConnMem ID 4: after 30s: 8.426MB slope30-300: 0.003MB/s slope300+ 0.001MB/s
ConnMem ID 6: after 30s: 1.855MB slope30-300: 0.000MB/s slope300+ -0.000MB/s
ConnMem ID 7: after 30s: 3.385MB slope30-300: -0.000MB/s slope300+ 0.000MB/s
ConnMem ID 11: after 30s: 0.000MB slope30-300: 0.000MB/s slope300+ 0.001MB/s
ConnMem ID 13: after 30s: 0.000MB slope30-300: 0.035MB/s slope300+ 0.001MB/s
ConnMem ID 14: after 30s: 3.025MB slope30-300: 0.001MB/s slope300+ 0.000MB/s
ConnMem ID 15: after 30s: 4.420MB slope30-300: 0.122MB/s slope300+ 0.090MB/s
ConnMem ID 16: after 30s: 3.835MB slope30-300: 0.003MB/s slope300+ 0.001MB/s
Memory budget (after 30s): 451.6MB
Attempt: 225.8MB => 873s, Inactivity: 225.8MB, 1653s
generating bro config /tmp/2006-05-23-broconf/2006-05-23_16-09_config-0.bro
```

Figure 4.23: Output of our tool after the systematic measurement phase

that script memory usage does not scale linearly with the number of random sampled connections. In addition, the configuration with ID 10 (http-reply) exceeds the CPU time budget all alone. The tool therefore assumes, that this configuration does not come into consideration to be combined with others.

Next, the tool computes, that the combination of the remaining analyzers exceeds the given CPU time limits. It therefore asks the operator to choose an analyzer to deactivate. In this example the analyzer with the ID 9, that imposes the highest additional load, is chosen. After deactivation, the CPU usage of the remaining configuration combination is reevaluated. Since it now adheres to the CPU time limits, the memory usage is computed. Note that the numbers for connection state memory usage are rather inaccurate for all but the baseline configuration. The tool now subtracts from the overall user specified core-memory budget (500 MB) the memory usage after 30 seconds and devotes the remaining budget half to connection attempts and to established but not terminated connections by determining the connection timeouts accordingly.

Finally, the tool outputs a bro policy script file that loads all analyzers that were not deactivated before. It also includes the determined connection timeouts.

4.5.5 Limitations

As we have seen in the end of Section 4.5.3, we are able to derive timeouts from the memory footprints of Bro running the BroBase configuration. However, deriving connection state timeouts for more complex configurations raises problems. For the moment we only approximate additional per connection state held in the event-engine. This is a purely technical problem and can be fixed by refining the Bro-internal instrumentation to accurately account for per analyzer state.

The problem of user defined state that is not associated to single connections is more general. At the moment the tool only flags a non-linear dependency between overall memory consumption and random sampled connections. The problem is, that to adapt our model of Bro memory consumption as discussed in Section 4.4.6, in general we need to understand how analyzers associate user-state to traffic. As an alternative to refining the model, we think it is possible, to automatically derive functions for these analyzers, that describe how the memory consumption of user state scales with random connection sampling. These functions could be derived from running the affected analyzers with different subsampled traces and analyzing the memory usage of these runs. As result, the memory consumption for user state in the unsampled traffic can be extrapolated without (manually) analyzing the user-defined code. This would be the first step on the way to automatically derive expire timeouts for large user defined data structures.

Our approach for extrapolating the real CPU time from the connection-sampled trace depends, as discussed earlier, on the assumption that each connection adds the same amount of workload as any other connection of the same type. We have seen in Section 4.4.6 an example, where a single outlier connection caused a substantially higher workload than all other connections. The POP3 analyzer in the version of Bro that we used is not optimized to parse large connections efficiently. In general, our experience shows, that large deviations from the average analysis time for a connection usually come

from coding mistakes (often rather an implicit presumption of a traffic property). These show an impact only for single (often bogus) connections that generate more workload than all others. If such a connection is included in the randomly sampled set of connections, then its above average contribution is multiplied by the connection sampling factor, resulting in a much too high load. Besides the POP3 analyzer, we found Bro's FTP analyzer to be in severe trouble on analyzing non-ftp connections: if it got to analyze a binary (non FTP) connection, it spent too many CPU cycles on every binary packet by trying to coerce it into FTP semantics. Although we fixed the problem for this special case, we cannot prevent a user from programming his own script and introducing a similar problem by mistake or even by deliberately choosing an expensive analysis algorithm.

Last our methodology so far determines a configuration for Bro that is able to cope with the traffic as recorded in the trace. However, using this 20 minutes sample we cannot anticipate how the traffic changes during a longer period of time. Particularly network traffic is subject to strong time-of-day and time-of-week effects. If the analysis trace was captured at 4am in the morning we will get a configuration that likely causes an overload during the time of day when network load is often by a factor of three higher. In the next Section, we present an approach to overcome this limitation and to predict the resource usage over a longer time based on our 20 minutes traffic sample.

4.6 Prediction of Long-Term Resource Usage

Our idea here is to use more summarized traffic measurements for predicting the resource usage of our NIDS based on the measurements described in Section 4.5. One reason, that a 20 minute sample cannot be enough to assess the resource usage of a network monitoring system is that the intensity of network traffic is highly dependent on the local time at which the network is monitored: Usually the traffic pattern exhibits strong time-of-day and day-of-week effects. That means that one expects, e.g., in a local company's network on the night between Saturday and Sunday, at 4am much less traffic than on Monday morning at 11am.

We confirmed that CPU and memory usage for Bro mainly scale with the number of processed connections. Here, we rely on network data aggregated to the connection level to predict the resource usage of the NIDS. Therefore, we correlate the number of processed connections during a short measurement with the number of connections in the connection level data. Examples of this data are Bro connection logs or netflow data as it is generated by routers directly.

4.6.1 Idea

Once we can predict the resource usage of the NIDS depending on the time, we can come up with schemes to vary the configuration depending on the local time. The goal of adapting the NIDS configuration over time is to effect that the system is not overloaded during times of high traffic throughput and not underloaded during times with less traffic (see Section 3.7.3). Another idea is to determine a partitioning of the traffic in order to

either load balance or prioritize the analysis: For example, in a network environment a popular web server could be excluded from the NIDS' analysis, with the argument, that the requests and responses are constantly monitored by the operators anyway.

Simple Projection Approach

For the basic connection analysis (that is the BroBase configuration) Bro's resource consumption scales linear with the number of connections. The number of connections per unit of time, in turn, varies due to time-of-day effects. The simplest approach for predicting Bro's resource usage over longer time intervals is to compare the number of connections analyzed during the measurements with the number of connections per unit of time as determined by connection-level data.

For "projecting" the CPU time needed for Bro running the BroBase configuration, we can thus apply the rule of proportion: If in our short measurement the analysis of N connections took P sec. of CPU processing time, we predict a Bro instance performing the same analysis for N/R connections to use P/R sec. of CPU time. For memory usage we need a slightly different approach. In contrast to CPU time, memory usage cannot be measured per time interval. The number of connections analyzed in a time interval does not directly imply the memory consumption after that interval: Some of the connections are already terminated (thus not occupying any memory), others are still active. Memory usage at any moment depends on how many connections are active at that moment. However, predicting the number of connections in state at a time T enables us to apply the rule of proportion in a similar way as for the CPU time: From our short measurement we know the memory usage of X connections, so we can calculate the total memory usage for the number of connections predicted at time T.

Projection Approach for Complex Configurations

Clearly, for the simple projection approach to work, we have to assume, that all connections are analyzed in the same manner, independent on when they took place. However we learned from Sections 3.6.4 and 4.3 that not all connections are treated equally. Most obviously, the resource usage for a connection depends on how deep it is analyzed. For predicting the influence that changes in the traffic composition have on resource usage, we have to identify a set of connection parameters that *(i)* influence Bro's resource usage per connection and *(ii)* that are not uniformly distributed across connections over time. If all parameters would be distributed uniformly over the connections (meaning the traffic mix would stay the same, no matter how many connections pass), we could just follow the simple approach outlined above and directly scale resource usage with the number of connections per time interval: The number of connections in the single "connection classes" that are analyzed differently would for each class scale linearly with the total number of connections.

According to our decomposition of Bro's analysis work (see Section 4.3) and our experience from the systematic measurements in Section 4.4 we split the number of connections per time unit along the following, independent "dimensions". Each of these dimensions

can be directly derived from Bro connection summaries.

- connection state
- service / analyzer
- connection duration

The connection state determines whether the connection could potentially be analyzed by any application layer protocol analyzer. A connection that has not been properly established (i.e., one that only consists of a single SYN request) is handled by Bro's connection compressor. These connections occupy much less memory resources than successfully established connections, since they are not analyzed further. Fully established connections however, are modeled in Bro by rather large in-memory objects encapsulating all state needed for successful protocol decoding. The percentage of unsuccessful connections is expected to vary over time, therefore we have to distinguish these two classes of connections for our resource usage prediction.

Each protocol analyzer is realized independent of the others, each one performs different tasks, and we have seen, that each needs different resources. At the same time, the application mix that a network environment exhibits, shows, just as the overall number of connections, often strong time-of-day and day-of-week dependencies: In most network environments, some bulk transfers and robots define the application mix in the off-hours. During the busy hours however, the traffic mix is usually dominated by the individual user behavior. Both, the off-hour and the busy hour mixes strongly dependent on the user community and the policy of the network environment.

The connection duration is important for determining the number of active connections in state at each point in time. Longer connections do not necessarily need more memory than short ones, but the memory for those connections is not released as quickly as for short connections. As with the other dimensions, the exact composition of the connection-durations changes dependent on the local time: Typically during nighttime longer lasting bulk transfers contribute a larger fraction of the total number of connections than during daytime.

In addition to these dimensions that influence the resource consumption of a NIDS directly, it is also possible to add a parameter class, *direction* or *source and destination*. These are dimensions that come into question to the operator for prioritizing: He might be interested in just analyzing connections originating outside his local network or he might want to not analyze connections directed towards a popular web server. Obviously diurnal variation in the traffic mix to and from different parts of the network result in changes in the resource consumption of the monitoring system too.

None of our dimensions described here is dependent on the volume of bytes transferred in a connection. The volume of a connection often influences the resource consumption for its analysis (after all large connections need more CPU time to be parsed than small ones). However, for our projection approach, we do not need to introduce a dimension for connection volume if the distribution of connection volumes does not show significant variation. In this thesis, we restrict our model to not take connection volume

Input:

`conn_level_data`	Connection level data including the following information:
`conn_start`	Connection start time
`conn_service`	Port or application layer protocol of the connection
`conn_state`	Information on if and how the connection was terminated
`conn_dur`	Duration of the connection

```
foreach conn in conn_level_data
{
   dur_class = int(log10(conn_dur))
   agg_conn_data[timebin(conn_start)][conn_service][conn_state][dur_class]++
}
```

Output: `agg_conn_data`

Figure 4.24: Aggregation of connection level data for long-term resource prediction

distribution changes (in the overall traffic and within the single classes) into account. In the future, the model can be refined by adding a dimension for the transferred volume of the connections.

4.6.2 Long-Term Resource Usage Projection

Based on the dimensions identified above, we in a first step aggregate the connection level data in time bins along these dimensions. That means, for each time bin we fill a three-dimensional table with the number of connections per service, per connection state, and per connection duration (see Figure 4.24 for a pseudo-code description). The size of the data structure needed to hold this table is a function of the number of dimensions and the size of dimensions. Basically, the data structure grows exponentially with the number of dimensions. Additionally, it depends directly on how many classes there are along one dimension: Therefore the number of dimensions and the number of classes in the individual dimensions have to be kept reasonably small (in the order of 10s).

Following our decomposition of the analysis work of Bro in Section 4.3 and Section 4.4, we implement the three dimensions listed above in our prototype as follows: The dimension connection state has five classes (attempt, established, half-closed, reset, closed) and the dimension service has 40 classes (one for each Bro application layer protocol analyzer and a few other well known service ports). The connection duration is not readily divided into discrete classes. To reduce the number of classes for connection duration to a manageable number, we group the connection durations into classes using the first integer number smaller than the logarithm to the base of ten. The class C of a connection with duration D seconds is thus determined as $C = \lfloor log_{10} D \rfloor$.

Input:

`agg_conn_data`	Aggregated long term connection level data (see Figure 4.24)
`sum_conn_sample`	Summarized (according to Figure 4.24 with only one time-bin) connection level data of a sample measurement (see Section 4.5.3)
`base_res_prof`	Bro resource usage profile for the BroBase configuration (see Figure 4.18)

```
avg_cpu_per_conn = sum_cpu_usage(base_res_prof) / sum_conn_sample[*][*][*]

foreach timebin in agg_conn_data
{
   cpu_usage[timebin] = agg_conn_data[timebin][*][*][*] * avg_cpu_per_conn
}
```

Output: `cpu_usage` (per timebin)

Figure 4.25: Pseudo-code simple projection approach for CPU usage

Simple CPU Time Projection

Now we apply our methodology outlined in Section 4.6.1 to predict CPU time based on the summarized traffic data and our short measurement sample as described in Section 4.5.3. First, we use the simple approach for the prediction of CPU time of the basic BroBase configuration. We do not have to separate the connections along our dimensions, as we for the BroBase configuration assume that they are all treated approximately the same[4]. We quantify the CPU time needed for processing a single connection by computing the average CPU time requirement for one connection from our base-measurement as described in Section 4.5.3. This average value is multiplied with the total number of connections to process for each time bin. Figure 4.25 describes our approach for simple CPU time projection as pseudo-code.

In Figure 4.26 we show, for the basic Bro configuration BroBase, a comparison between the measured CPU time and the predicted CPU time based on Bro connection summaries. The average CPU time per connection is computed from a sample 20 minutes dataset (`May2006-20min`) captured and analyzed on May, 23 using our methodology described in Section 4.5.3. We note that overall the trend of the predicted CPU time matches the trend of the measured CPU time closely. The approach even accounts correctly for the outliers: At all times there is an outlier in the measured data we do also see an outlier in the predicted CPU times. In general the predicted CPU times are somewhat lower than the measured ones but at the same time they show less fluctuation. The greater fluctuation for the measured samples is in part due to the higher resolution

[4] Actually the connection compressor imposes slightly less workload on unsuccessful connection attempts than the ordinary analysis does on fully established connections. However, our instrumentation for CPU time usage cannot attribute CPU time to these individual classes of connections

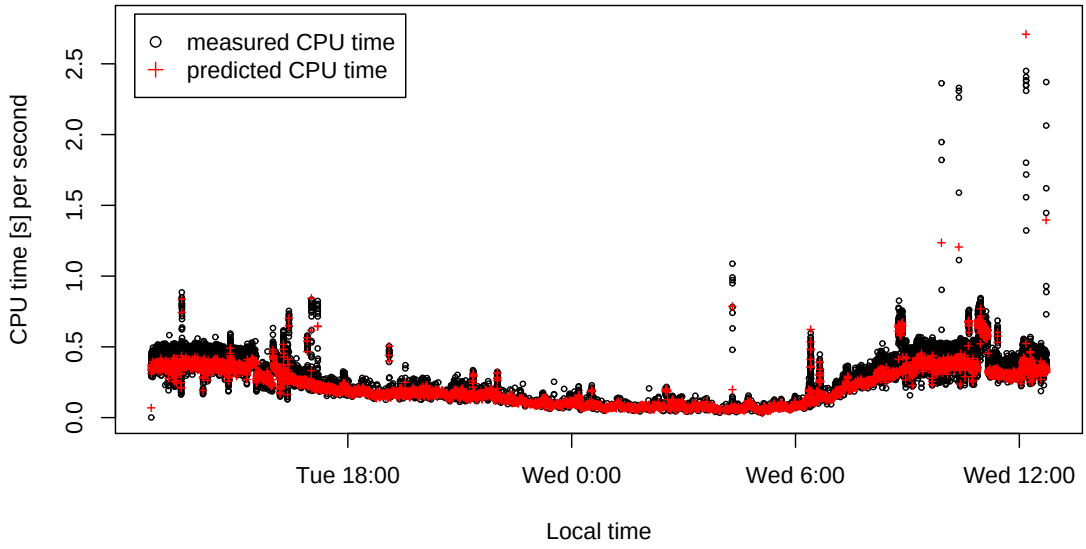


Figure 4.26: BroBase configuration: Measured CPU time vs. predicted CPU time for trace `mwn-full-packets`

of the dataset: The measurements are taken every second, but the prediction refers to the average over ten second time bins. In the scatter plot in Figure 4.27 we directly compare the CPU time averaged over ten seconds with the predicted average CPU time for each ten second time bin. We note that there are only very few outliers. In general, our prediction matches the measurement well, yet it tends to underestimate the actual work. The mean absolute error of this underestimation is -0.022 sec. CPU time per second, the mean relative error is -8.0%.

The results presented in Figure 4.26 confirm again, that for the BroBase configuration in principle CPU usage scales linearly with the number of processed connections. Another experiment with the 24h trace `mwn-full-packets` shows, that the time that is consumed for processing each connection is not necessarily constant: We let Bro analyze the `mwn-full-packets` trace again with the BroBase configuration but disable the inactivity timeout. The CPU time used per second is shown in the black points in Figure 4.28. Although all data structures involved in connection handling are implemented so that they have amortized constant access times, in Figure 4.28 we see that the processing time per connection slowly ramps up. The reason is that the data structures holding the connection state grow steadily as not terminated connections are accumulated. The red points denote again our predicted time as in Figure 4.26. In contrast to the measurement, our prediction scales the workload per connection "state-less". This hints at another source of errors for our prediction: So far we consider the mean CPU time consumed for a connection to be independent of the parameters that control how much state is kept. In our measurement methodology (see Section 4.5.3) we chose rather conservative state timeouts for connections; i.e., the inactivity timeout is disabled. Al-

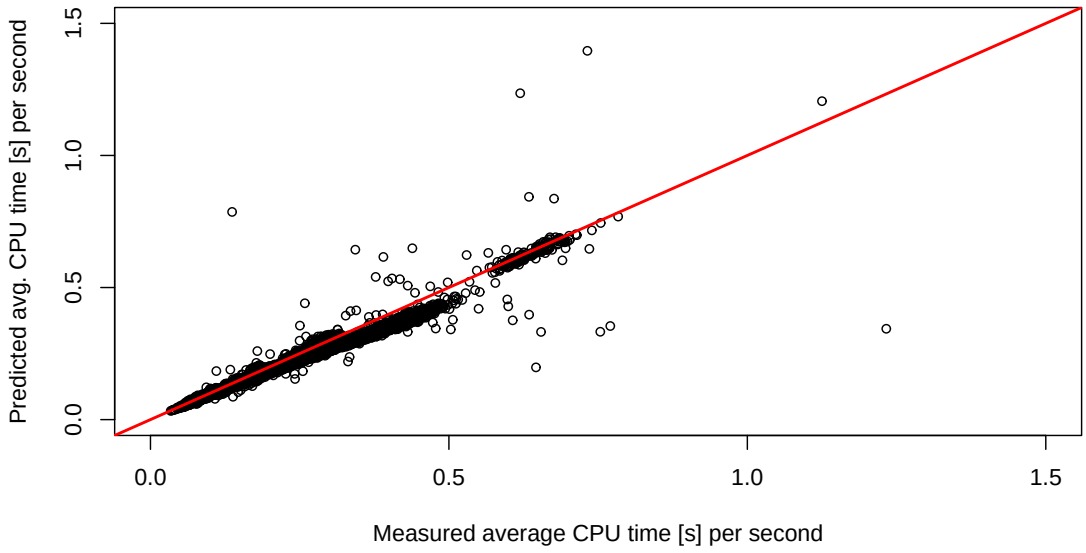


Figure 4.27: BroBase configuration: Scatter plot measured CPU time vs. predicted CPU time for trace `mwn-full-packets`

though the small sample does only allow state to be collected for the total time of the sample (20 minutes) the size of the data structure may already influence the average CPU time per processed connection.

CPU Time Projection for Complex Configurations

For more complex configurations (i.e., configurations with more analyzers loaded), we use our connection counts separated into classes according to our three dimensions to predict CPU time based on the connection level data. The result allows an NIDS operator to easily compare different configurations of the same NIDS in his network environment. For example he can use the long term projection of CPU usage to derive a set of appropriate configurations for our load-level concept (see 3.7.3).

For the prediction of CPU time we need the dimensions *service* and *connection state*. We define connection classes for each combination of a service and a connection state. Thus, each class contains connections that may contribute (depending on the configuration of Bro) a different CPU workload. The idea is, that each analyzer's workload scales with the amount of connections belonging to one or more classes. The FTP analyzer for example in its default configuration contributes workload only for successfully established FTP connections. Therefore, we first quantify the average CPU time usage the FTP analyzer imposes for one single successfully established connection with the service "FTP" in our base-measurement (note that the connection size distribution of FTP connections is included in that number). Then we multiply this number with the number of successfully established FTP connections present in our aggregated data for

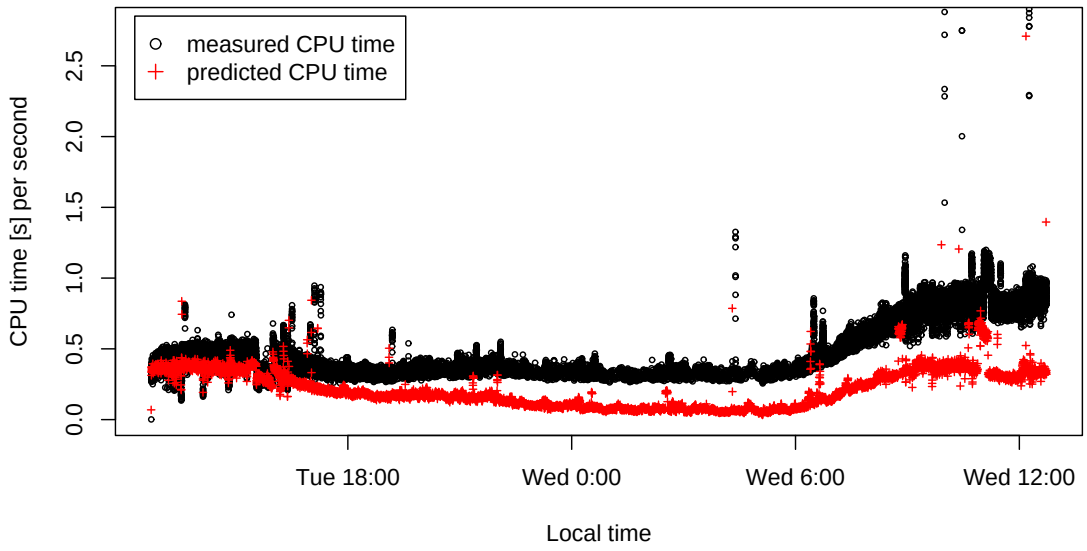


Figure 4.28: measured CPU time with long timeouts vs. predicted CPU time for trace `mwn-full-packets`

each time bin. For other analyzers we use the same methodology: For each we identify the connection classes that are analyzed and multiply the number of connections in this category with the average CPU time needed to analyze a single connection. Figure 4.29 summarizes our methodology for predicting the CPU usage of single analyzers as pseudo-code. In the end, we can sum up the CPU times for single analyzers. Thus, we get the CPU time usage per time bin for any configuration (as a combination of the appropriate analyzers).

Of course, this methodology is only applicable, if for each connection class there is a sufficient number of connections in our base measurement. If there are no connections at all for a single class in the base measurement, we can not make a projection of the analysis cost, if connections of that same type appear in the long-term connection data. The same problem persists, if we have only a small number of connections (e.g., only one) in the base measurement for a single connection class. According to our model, in an extreme case, we make one connection responsible for the additional workload imposed by a single analyzer. Since we cannot measure the exact CPU time needed for analyzing that single connection, we do not have reliable numbers for the additional CPU time per connection needed for that analyzer. As long as connections of this class do not occur often in the long-term observation we do not need exact numbers. It is tempting to argue that this is likely the case: after all there are only a few connections of the class observed during the basic measurement. However, this can indeed be a problem. For example during our experiments we found that ssh and telnet connections did not occur frequently in the short measurement trace `May2006-20min`. Yet in the long-term connection trace surges of (short) connections are observable, likely due to

Input:

`agg_conn_data`	Aggregated long term connection level data (see Figure 4.24)
`sum_conn_sample`	Summarized (according to Figure 4.24 with only one time-bin) connection level data of a sample measurement (see Section 4.5.3)
`config_list`	List of Bro configurations (analyzers) and the class of connections they analyze (User-specified)
`rel_res_prof_tab`	Table of relative resource profiles (one profile per config in `config_list`) (see Figure 4.18)

```
foreach config in config_list
{
   avg_cpu_usage_per_conn[config] =
      sum_cpu_usage(rel_res_prof_tab[config]) /
      sum_conn_sample[service(config)][states(config)][*]

   foreach timebin in agg_conn_data
   {
      cpu_usage[config][timebin] =
         agg_conn_data[timebin][service(config)][states(config)][*] *
         avg_cpu_usage_per_conn[config]
   }
}
```

Output: `cpu_usage` (per config and per timebin)

Figure 4.29: Pseudo-code CPU projection approach for complex configurations

brute-force login attempts. Consequently, we cannot reliably extrapolate the analysis cost of these surges from only a small number of connections in our base measurement. For the implementation of our methodology, we do not extrapolate CPU time usage for connection classes that included less than one percent of all connections in our base measurement. If the long-term connection level data contains connections of that class, a warning is issued and the connection class is ignored.

In Figure 4.30 and Figure 4.31 we show the results for the prediction of CPU time for a complex Bro configuration. We run a Bro instance configured to run 12 analyzers (finger, frag, ftp, http-request, ident, irc, login, pop3, portmapper, smtp, ssh, and tftp) in addition to the BroBase configuration. The CPU time is predicted by applying our projection approach subsequently to all analyzers and the respective connection classes. The plotted result is the sum of the predicted CPU time for the basic configuration and the predicted additional CPU usage of each analyzer.

In Figure 4.30 we see, that overall our prediction matches the measurement. Note that the value range (y-axis) of the measurement is quite large. The outliers are not always predicted, in part again attributable to the higher resolution of the measurement (1 sec. in the measurement vs. 10 sec. in the prediction). We also note, that our approach

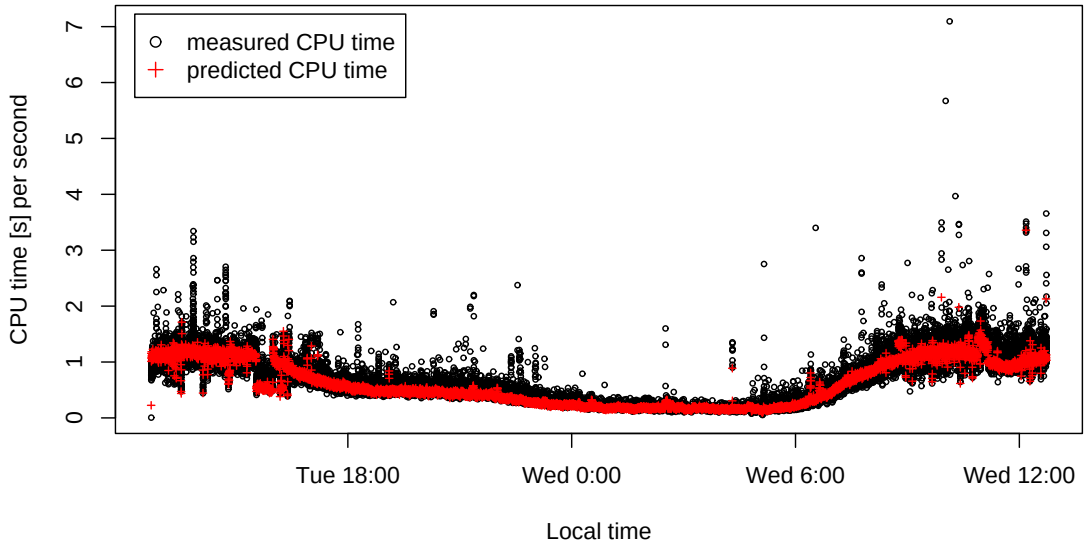


Figure 4.30: Complex configuration: Measured CPU time vs. predicted CPU time for trace `mwn-full-packets`

to only extrapolate CPU time for connections that contribute a significant portion of connections in our base measurement results in a underestimation of CPU time in the case that suddenly many of these connections need to be analyzed.

In Figure 4.31 we compare the predicted CPU time and the measured CPU time averaged over 10 second-bins directly. Comparing the averaged measurements eliminates the error of the higher measurement resolution. Despite this, we note, that most of the outliers are underestimated. The mean absolute error of our prediction approach for the used Bro configuration is -0.029 sec. CPU time per second traffic and the mean relative error is -4.6%. The reason why the mean relative error is even smaller than for the prediction of the CPU time needed for the BroBase configuration is that in general the absolute numbers of the measured samples are much larger for our complex configuration.

Memory Usage Projection

The basic idea for predicting the memory usage is to derive the number of active (in-state) connections at any moment in time from the connection-level data. Once we can project the memory usage for a given configuration, an operator can, e.g., compare different settings of the connection expire timers for the network traffic in his environment.

In our prototype we focus on predicting the number of TCP connections in memory at the end of each bin of our aggregated connection level data. In a second step, this number can be multiplied with the average memory consumption per connection of the connection type in question. As our instrumentation of Bro does not give us accurate

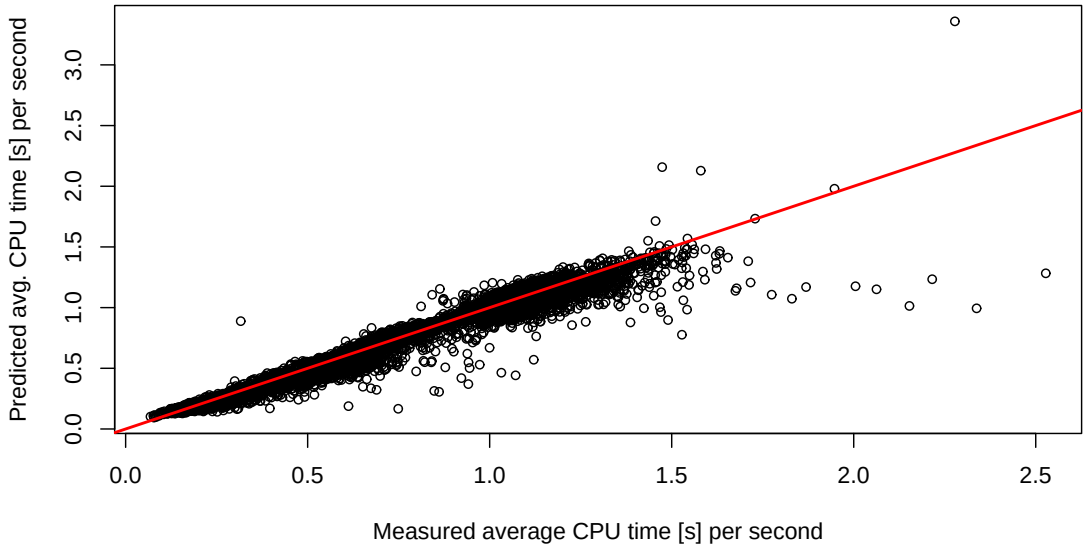


Figure 4.31: Complex configuration: Scatter plot measured CPU time vs. predicted CPU time for trace `mwn-full-packets`

numbers for additional memory used for connections that are analyzed by a special (application layer protocol) analyzer, we do not aim at predicting the exact amount of memory used for Bro's internal connection state. Furthermore, our prototype currently ignores script level state. For some scripts, similar techniques to the one described here can be used to predict the number of connections active in user-level state but in general it is very difficult to exactly model the dependencies between active connections in core state and those in user state.

For predicting the number of active connections in Bro's memory, we primarily use the classification of our connection-level data along the dimensions *connection duration* and the final *connection state* of the connection. In fact, for each connection, memory for its state is occupied for the lifetime of the connection plus an appropriate connection expire timer. To determine what connection timer in Bro is responsible for expiring a connection from state, we use the final connection state. Our prototype currently distinguishes four types of connections: Connection attempts, connections that are terminated normally, connections that are reset and connections that are not- or only half-terminated. For each of these types Bro features a separate connection expire timer.

Thus, from the connection duration and the responsible timeout for a connection class, we can determine a timespan T that these connections occupy memory as they are held in state. Remember, that T is actually a low estimate for this timespan, as our classification of connection duration groups connections following the formula $C = \lfloor log_{10} D \rfloor$ (where C denotes the connection class and D the connection duration). If T for a connection class is longer than the bin size B of our aggregated connection-level data, these connections are kept in memory at least for T/B bins. In our implementation, we address this by

keeping track of the number of connections that have to be expired for each class at any point in time.

However, with bin sizes around tens of seconds, usually most regular connections (plus their connection expire timeout) are shorter than the bin size. Still, at an arbitrary point in time a number of these connections are held in state. For these short-living connections (e.g., standard HTTP connections originating from interactive web surfing) we follow a similar approach as for the prediction of CPU time. For both our base measurement and the current connection bin, we determine the number of these connections per second network traffic. For our base measurement we additionally determine the number for active short living connections (N_X) at one point in time X: We count the number of successfully established and terminated connections that are shorter than our bin size and that start before and end after time X. Let the number of short-living connections per second be S_{base} for the base measurement and S_{bin} for the current time bin. Then we multiply the ratio of S_{bin} and S_{base} (S_{bin}/S_{base}) with N_X in order to extrapolate the number of short-living active connections at the end of the current bin. The pseudo-code in Figure 4.32 summarizes our methodology for predicting the number of connections to be kept in state for configurable connection expiration timeouts.

Unfortunately, for certain connection types, the connection duration in our connection-level data does depend on the semantics of a connection in the data. With our prototype, we rely on Bro connection summaries as high level data. Thus the duration field of the connections in the connection log depends on the configuration of the connection timeouts for the Bro instance that was used to collect that data. For example assume, that this instance was configured to run the BroBase configuration with an *inactivity timeout* of 300 sec. With that configuration Bro only analyzes connection control packets. As a consequence, the connection log does not show any connection longer than 300 sec since connections are expired by the inactivity timeout if the time between connection handshake and teardown is longer than 300 sec. For a connection that lasts longer than 300 sec, the connection log contains a connection in final state "*established but not closed*" and an additional connection record in state "*only teardown seen*". This implies the connection log contains more connections if shorter connection timeouts are used.

For the number of concurrently active connections, we carefully keep book of the number of connection in state for each class of connections separately. In each class, the dimension *connection state* clearly influences how much memory a connection consumes: Remember, that with the connection compressor connection attempts need drastically less memory than fully instantiated connections. The dimension *service/analyzer* classifies connections according to different additional connection state introduced by single Bro analyzers. Multiplying the average size of a connection in each connection class with the number of connections in that class allows to compute the amount of memory consumed by that class of connections for every time bin.

Due to the lack of more accurate instrumentation of the core memory use of the Bro application protocol analyzers our prototype cannot compute the exact memory usage of the connections held in state. Currently, our prototype computes only numbers for the average memory consumption of connection attempts and for fully established connections in general.

Input:

`agg_conn_data`	Aggregated long term connection level data (see Figure 4.24)
`N_X`	The number of active connections in the sample measurement at an arbitrary point in time (X)
`S_base`	The number of connections in the sample measurement with `conn_dur` < binsize
`config_list`:	List of Bro configurations (analyzers) and the class of connections they analyze (User-specified)
`state_to_timeout`	Table associating final connection states (in `agg_conn_data`) to connection timeouts for prediction (User-specified)

```
foreach timebin in agg_conn_data
{
  foreach config in config_list
  {
    #short living connections:
    S_bin = 0
    foreach (dur_class, state) where (contained(in=states(config), state) and
             dur_class + state_to_timeout(state) < binsize)
    {
      S_bin += agg_conn_data[timebin][service(config)][state][dur_class]
    }
    active_conns[config][timebin] = (S_bin / S_base) * N_X

    #long-living connections:
    foreach (dur_class, state) where (contained(in=states(config), state) and
             dur_class + state_to_timeout(state) >= binsize)
    {
      conn_contrib = agg_conn_data[timebin][service(config)][state][dur_class]
      long_active_conns[config][timebin] += conn_contrib
      expire_conns[config][timebin+10^(dur_class)+state_to_timeout[state]] +=
        conn_contrib
    }

    #expire old connections from accumulator (long_active_conns) and add
    #remaining long-living connections to current time bin:
    long_active_conns[config] -= expire_conns[config][timebin]
    active_conns[config][timebin] += long_active_conns[config]
  }
}
```

Output: `active_conns` (per config and per timebin)

Figure 4.32: Pseudo-code for predicting the number of connections in state

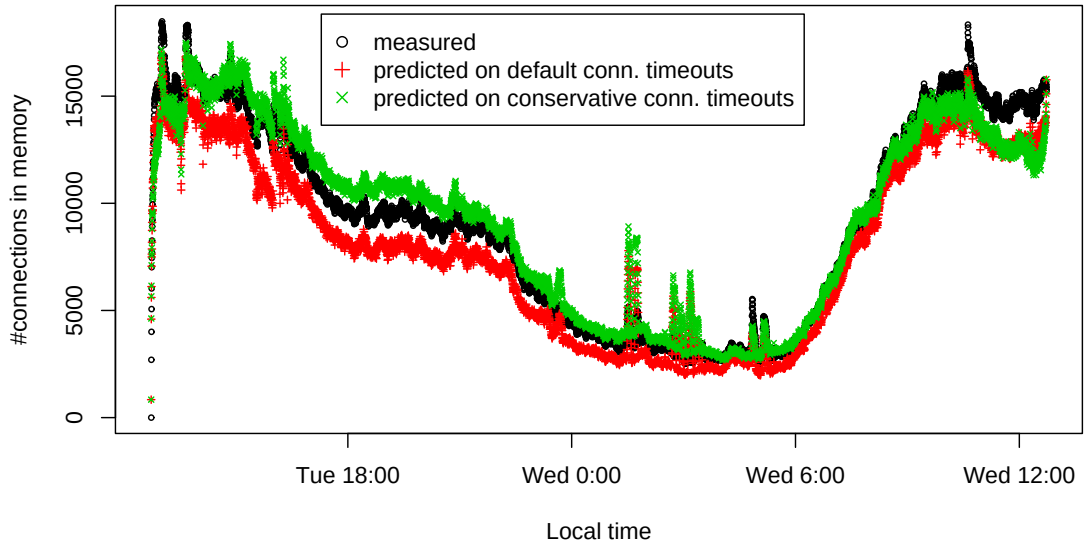


Figure 4.33: Predicted number of established connections in memory for trace `mwn-full-packets` and inactivity timeout 300 sec. (default)

In Figure 4.33 we show the results of our prediction for the number of full established connections in memory. We take the actual (measured) number of in-state connections from Bro's internal instrumentation, which is configured to output the number of connections in memory for each connection state separately. For the measurement Bro was configured to run the BroBase configuration with default connection timeouts (i.e. 300 sec. inactivity timeout) on the `mwn-full-packets` trace. The base measurement for the prediction is again the `May2006-20min` dataset. The two curves for the predicted numbers of active connections are based on connection logs from Bro instances configured with the same timeouts as the measurement instance (red points) and a Bro instance with a disabled inactivity timeout (green points). Apparently, in both cases the prediction approximately matches the measured number of active connections. However, we note, that for the connection-log version with disabled inactivity timeout (green points) the error is smaller: The mean relative error is +5.0% for the prediction based on the connection-log with disabled inactivity timeout and -13.7% for the prediction based on the connection-log with the 300 sec. inactivity timeout.

In a subsequent experiment, we measure the in-state connections of a Bro instance running on the `mwn-full-packets` trace with a disabled inactivity timeout. Remember, that this instance cumulates all connections in memory that are not properly terminated. Figure 4.34 shows, that the discrepancy between the predictions based on the different connection logs is striking. Apparently the prediction based on the connection log gained from a Bro instance with disabled inactivity timeout resembles the measured number of connections in memory well. On the other hand, the prediction based on the connection log originating from the Bro instance with default timeouts (300 sec. inactivity timeout)

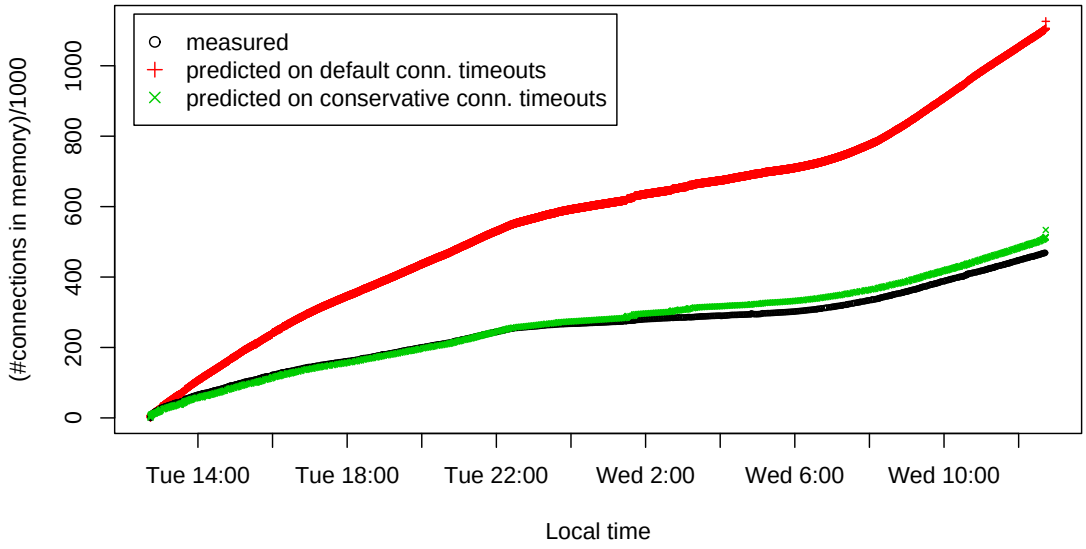


Figure 4.34: Predicted number of established connections in memory for trace `mwn-full-packets` and disabled inactivity timeout

grossly overestimates the actual number of active connections. The reason for this are the connections that are longer than the inactivity timeout. The connection log contains those in state "*established but not closed*" rather than "*established and terminated normally*" as they were expired before their regular connection termination by the inactivity timeout. This is counter-intuitive, as very few connections are supposed to be longer than 300 sec. Inspection of the connection level data shows, that this is indeed the case: Only 0.5% of all connections during the 24h of traffic we analyze last longer than 300 sec. in the sense that they are properly terminated after being active for at least 300 sec. But with a total of ca. 134 million connection entries in the connection log, that 0.5% resolve to the respectable absolute number of 695,000 connections, which makes up for the difference in Figure 4.34.

Overall, our experiments with predicting the number of active connections in memory shows, that with our prototype it is possible to run through what-if scenarios for different values of the connection timeouts. On the other hand, we note, that the prediction is highly susceptible to differences in the connection-level data. In particular the last experiment confirms the fundamental problem, that our prediction depends on the semantics of a connection in the connection-level data. For instance, our methodology without further heuristics is not suitable to predict the number of connections in memory if larger connection timeouts are supposed to be predicted than those that were used when collecting the connection-level data.

4.7 Conclusion and Future Work

Evaluating or deploying a NIDS in a network environment is a difficult and time consuming task for a network operator. Yet it is unlikely that operators know everything about their traffic either. Anyway, a network operator has to find a configuration of the system's parameters, that tunes the tradeoff between resource usage and detection capability according to his intention. A question that is always of interest in this context is: Given a set of parameters, what options does one have to tune the parameters, without exceeding the available machine resources (CPU and memory).

In this chapter, we present our approach for a tool that automatically compares different (default or user defined) configurations for a NIDS with regard to the resource consumption. We base our systematic measurements on full packet traces. Using our instrumentation of the NIDS, we show that we can directly compare the CPU usage of the NIDS running on these traces with the CPU usage of NIDS instances running on live network traffic. Thus we can decide, whether a specific NIDS configuration overloads the machine regarding needed CPU time. Since in high-volume environments it is not feasible to record full packet traces, we use random connection-sampled full packet traces. While the use of random sampling allows us to drastically reduce the disk space and the time that is needed for performing the systematic measurements, it imposes technical and conceptual difficulties.

Our tool that helps to compare the resource usage of different configurations of a NIDS does only analyze a very short timespan of the network traffic. Though it is possible to configure the tool to collect a much longer packet trace (by providing it with a very large disk space budget) this also results in unacceptable long measurement times. Therefore, we present an approach to project the CPU usage numbers gathered from the short systematic measurements onto a higher level representation of the network traffic. As the random connection sampling, this approach leverages, that the NIDS we model is connection oriented and therefore its resource usage in general scales linearly with the number of processed connections.

Our methodology for projecting CPU usage measurements onto a higher representation of long-term traffic is not limited to actually measured traffic. Given a guess or knowledge on how the traffic characteristics evolve, our methodology can be utilized to extrapolate the resource usage for a suitably crafted connection-level summary of the future traffic.

Especially refining the extrapolation of the memory usage from random connection-sampled packet traces, is an area of future work. For this Bro's internal memory accounting instrumentation has to be improved, to account for the additional state introduced by the optional analyzers. Furthermore, it should be worthwhile to automatically model the memory usage for user-level state dependent on the processed connections or similar metrics.

In this work, we evaluate our methodology only with traces from the MWN environment. Although the traffic observed in the MWN exhibits a lot of variation, it would still be valuable to systematically evaluate and refine our methodology and tools in other network environments.

In summary, our work provides us with a detailed understanding of the dependencies between different traffic characteristics and the resource usage of a stateful NIDS. The resulting model is the basis for our tool that allows an operator to elicit what options for the parametrization of the NIDS he has and for our methodology to extrapolate resource usage from short measurements and connection-level data.

5 Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection

5.1 Introduction

Recall, that network intrusion detection systems analyze streams of network packets in order to detect attacks and, more generally, to detect violations of a site's security policy. NIDSs often rely on protocol-specific analyzers to extract the higher-level semantic context associated with a traffic stream, in order to form more reliable decisions about if and when to raise an alarm [SP03b]. Such analysis can be quite sophisticated, such as pairing up a stream of replies with previously pipelined requests, or extracting the specific control parameters and data items associated with a transaction.

To select the correct analyzer for some traffic, a NIDS faces the challenge of determining which protocol is in use before it even has a chance to inspect the packet stream. To date, NIDSs have resolved this difficulty by assuming use of a set of well-known ports, such as those assigned by IANA [IAN], or those widely used by convention. If, however, a connection does not use one of these recognized ports – or misappropriates the port designated for a different application – then the NIDS faces a quandary: how does it determine the correct analyzer? In Chapter 4, we have seen, that using the wrong analyzer for a connection is a waste of resources. Trying obstinately to coerce a binary stream into plain text-protocol syntax can even cause excessive load spikes.

In practice, servers indeed do not always use the port nominally associated with their application, either due to benign or malicious intent. Benign examples include users who run Web or FTP servers on alternate ports because they lack administrator privileges. Less benign, but not necessarily malicious, examples include users that run servers offering non-Web applications on port 80/tcp in order to circumvent their firewall. In fact, some recently emerging application-layer protocols are *designed* to work without any fixed port, primarily to penetrate firewalls and escape administrative control. A prominent example is the voice-over-IP application Skype [BS06], which puts significant efforts into escaping restrictive firewalls. Sometimes such applications leverage a common protocol and its well-known port, like HTTP, to *tunnel* their payload not just through the firewall but even through application layer proxies. In these cases, analyzing the application's traffic requires first analyzing and stripping off the outer protocol before the NIDS can comprehend the semantics of the inner protocol. Similarly, we know from operational experience that attackers can attempt to evade security monitoring by concealing their traffic on non-standard ports or on ports assigned to different protocols: trojans installed on compromised hosts often communicate on non-standard ports; many botnets use the IRC protocol on ports other than 666x/tcp; and pirates build

file-distribution networks using hidden FTP servers on ports other than 21/tcp.

It is therefore increasingly crucial to drive protocol-specific analysis using criteria other than ports. Indeed, a recent study [Som05] found that at a large university about 40% of the external traffic could not be classified by a port-based heuristic. For a NIDS, this huge amount of traffic is very interesting, as a primary reason for not using a standard port is to evade security and policy enforcement monitoring. Likewise, it is equally pressing to inspect whether traffic on standard ports indeed corresponds to the expected protocol. Thus, NIDSs need the capability of examining such traffic in-depth, including decapsulating an outer protocol layer in order to then examine the one tunneled inside it.

However, none of the NIDSs which are known to us, including Snort [Roe99], Bro [Pax99], Dragon [Ent], and IntruShield [Int], use any criteria other than ports for their protocol-specific analysis. As an initial concession to the problem, some systems ship with signatures – characteristic byte-level payload patterns – meant to *detect* the use of a protocol on a non-standard port. But all only report the mere fact of finding such a connection, rather than adapting their analysis to the dynamically detected application protocol. For example, none of these systems can extract URLs from HTTP sessions on ports other than the statically configured set of ports.[1] With regards to decapsulating tunnels, a few newer systems can handle special cases, e.g., McAfee's IntruShield system [Int] can unwrap the SSL-layer of HTTPS connections when provided with the server's private key. However, the decision that the payload *is* SSL is still based on the well-known port number of HTTPS.

The impetus for performing protocol analysis free of any assumptions regarding applications using standard ports arose from our operational experiences with NIDSs at our three large-scale network environments: the *University of California, Berkeley (UCB)*, the *Münchener Wissenschaftsnetz (Munich Scientific Network, MWN)* and the *Lawrence Berkeley National Laboratory (LBNL)* as described in Chapter 2. We found that increasingly significant portions of the traffic at these sites were not classifiable using well-known port numbers. Indeed, at UCB 40% of all packets fall into this category [Som05].

Being research environments, the three networks' security policies emphasize relatively unfettered connectivity. The border routers impose only a small set of firewall restrictions (e.g., closing ports exploited by major worms). MWN uses a more restrictive set of rules in order to close ports used by the major peer-to-peer (P2P) applications; however, since newer P2P applications circumvent such port-based blocking schemes, MWN is moving towards a dynamic traffic filtering/shaping system. In a first step it leverages NAT gateways [FBR06] used to provide Internet access to most student residences, and the IPPP2P system for detecting peer-to-peer traffic [IPP].

In this chapter we discuss the design, implementation, deployment, and evaluation of an extension to a NIDS to perform dynamic application-layer protocol analysis. For each connection, the system identifies the protocol in use and activates appropriate analyzers. We devise a general and flexible framework that (i) supports multiple ways to recognize

[1] To keep our terminology simple, we will refer to a single fixed port when often this can be extended to a fixed set of ports.

protocols, (ii) can enable multiple protocol analyzers in parallel, (iii) copes with incorrect classifications by disabling protocol analyzers, (iv) can pipeline analyzers to dynamically decapsulate tunnels, and (v) provides performance sufficient for high-speed analysis.

We demonstrate the power our enhancement provides with three examples: (i) *reliable* detection of applications not using their standard ports, (ii) payload inspection of FTP data transfers, and (iii) detection of IRC-based botnet clients and servers. The prototype system currently runs at the border of the University of California, Berkeley (UCB), the Münchener Wissenschaftsnetz (Munich Scientific Network, MWN), and the Lawrence Berkeley National Laboratory (LBNL). These deployments have already exposed a significant number of security incidents, and, due to its success, the staff of MWN has integrated bot-detection into its operations, using it for dynamic inline blocking of production traffic.

The remainder of this chapter is organized as follows: In Section 5.2 we analyze the potential of non-port-based protocol detection using the full packet trace `mwn-full-packets` and discuss the limitations of existing NIDSs. In Section 5.3 we present the design and implementation of our dynamic architecture and discuss the tradeoffs one faces in practice. Section 5.4 demonstrates the benefits of the dynamic architecture with three example applications. In Section 5.5 we evaluate the performance of our implementation in terms of CPU usage and detection capabilities. Finally in Section 5.6 we summarize our experience.

5.2 Analysis of the Problem Space

Users have a variety of reasons for providing servicing on non-standard ports. For example, a site's policy might require private services (such as a Web server) to run on an unprivileged, often non-standard, port. Such private servers frequently do not run continuously but pop up from time to time, in contrast to business-critical servers. From our operational experience, in open environments such servers are common and not viewed as any particular problem. However, compromised computers often also run servers on non-standard ports, for example to transfer sometimes large volumes of pirated content. Thus, some servers on non-standard port are benign, others are malicious; the question of how to treat these, and how to distinguish among them, must in part be answered by the site's security policy.

In addition, users also use standard ports for running applications other than those expected on the ports, for example to circumvent security or policy enforcement measures such as firewalls, with the most prevalent example being the use of port 80/tcp to run P2P nodes. A NIDS should therefore not assume that every connection on HTTP's well-known port is indeed a communication using the HTTP protocol; or, even if it *is* well-formed HTTP, that it reflects any sort of "Web" access. The same problem, although often unintentional and not malicious, exists for protocols such as IRC. These are not assigned a well-known privileged port but commonly use a set of well-known unprivileged ports. Since these ports are unprivileged, other applications, e.g., an FTP data-transfer connection, may happen to pick one of these ports. A NIDS therefore may encounter

traffic from a different application than the one the port number indicates. Accordingly the NIDS has to have a way to detect the application layer protocol actually present in order to perform application-specific protocol analysis.

5.2.1 Approaches to Application Detection

Besides using port numbers, two other basic approaches for identifying application protocols have been examined in the literature: (i) statistical analysis of the traffic within a connection, and (ii) locating protocol-specific byte patterns in the connection's payload.

Previous work has used an analysis of interpacket delays and packet size distribution to distinguish interactive applications like chat and remote-login from bulk-transfer applications such as file transfers [ZP00a]. In some particular contexts these techniques can yield good accuracy, for example to separate Web-chat from regular Web surfing [DWF03]. In general, these techniques [MZ05, RSSD04, KPF05, XZB05], based on statistical analysis and/or or machine learning components, have proven useful for classifying traffic into broad classes such as interactive, bulk transfer, streaming, or transactional. Other approaches model characteristics of individual protocols by means of decision trees [TC97] or neural networks [EBR03].

The second approach – using protocol-specific, byte-level payload patterns, or "signatures" – takes advantage of a popular misuse detection technique. Almost all virus-scanner and NIDSs incorporate signatures into their analysis of benign vs. malicious files or network streams. For protocol recognition, we can use such signatures to detect application-specific patterns, such as components of an HTTP request or an IRC login sequence. However, there is no guarantee that such a signature is comprehensive. If it fails to detect all instances of a given application, it exhibits *false negatives*. In addition, if it incorrectly attributes a connection to a given application, it exhibits *false positives*.

We can also combine these types of approaches, first using statistical methods (or manual inspection) to cluster connections, and then extracting signatures, perhaps via machine learning techniques [HSSW05]; or using statistical methods to identify some applications, and signatures to identify others [ZP00a] or to refine the classification, or to combine ports, content-signatures, and application-layer information [CKY+04].

In the context of NIDSs, signature-based approaches are particularly attractive because many NIDSs already provide an infrastructure for signature-matching (see Section 5.2.3), and often signatures yield tighter protocol identification capabilities.

5.2.2 Potential of a Signature Set

To evaluate how often common protocols use non-standard ports, and whether signatures appear capable of detecting such uses, we examine a 24-hour full trace of MWN's border router, `mwn-full-packets`. To do so we use the large, open source collection of application signatures included with the *l7-filter* system [L7]. To apply these signatures to our trace, we utilize the signature matching engine of the open source NIDS Bro [Pax99, SP03b]. Rather than running the l7-filter system itself, which is part of the Linux netfilter framework [NF], we convert the signatures into Bro's syntax, which gives

Port	Connections	% Conns.	Successful connections	% Success.	Payload [GB]	% Payload
80	97,106,281	70.82%	93,428,872	68.13	2,548.55	72.59
445	4,833,919	3.53%	8,398	0.01	0.01	0.00
443	3,206,369	2.34%	2,855,457	2.08	45.22	1.29
22	2,900,876	2.12%	2,395,394	1.75	59.91	1.71
25	2,533,617	1.85%	1,447,433	1.05	60.00	1.71
1042	2,281,780	1.66%	35	0.00	0.01	0.00
1433	1,451,734	1.06%	57	0.00	0.06	0.00
135	1,431,155	1.04%	62	0.00	0.00	0.00
< 1024	114,747,251	83.68%	101,097,769	73.73	2,775.15	79.05
≥ 1024	22,371,805	16.32%	5,604,377	4.08	735.62	20.95

Table 5.1: Ports accounting for more than 1% of the `mwn-full-packets` connections.

us the advantages of drawing upon Bro's built-in trace processing, connection-oriented analysis, and powerful signature-matching engine. We note however that while Bro and l7-filter perform the matching in a similar way, varying internal semantics can lead to slightly different results, as with any two matching engines [SP03b].

We begin by examining the breakdown of connections by the destination port seen in initial SYN packets. Table 5.1 shows all ports accounting for more than one percent of the connections. Note that for some ports the number of raw connections can be misleading due to the huge number of scanners and active worms, e.g., ports 445, 1042, and 1433. We consider a connection unsuccessful if it either does not complete an initial TCP handshake, or it does but does not transfer any payload. Clearly, we cannot identify the application used by such connections given no actual contents.

We make two observations. First, port-based protocol identification offers little assistance for most of the connections using unprivileged ports (totaling roughly 5.6 million connections). Second, the dominance of port 80 makes it highly attractive as a place for hiding connections using other applications. While an HTTP protocol analyzer might notice that such connections do not adhere to the HTTP protocol, we cannot expect that the analyzer will then go on to detect the protocol actually in use.

To judge if signatures can help improve application identification, for each of a number of popular apparent services (HTTP, IRC, FTP, and SMTP) we examined the proportion identified by the l7-filter signatures as indeed running that protocol. Table 5.2 shows that most of the successful connections trigger the expected signature match (thus, the signature quality is reasonable). Only for FTP we observe a higher percentage of false negatives. This can be improved using a better FTP signature. However, we also see that for each protocol we find matches for connections on unexpected ports, highlighting the need for closer inspection of their payload.

The differences in Table 5.2 do not necessarily all arise due to false negatives. Some may stem from connections without enough payload to accurately determine their protocol, or those that use a different protocol. Regarding the latter, Table 5.3 shows how often a different protocol appears on the standard ports of HTTP, IRC, FTP and SMTP.

Method	HTTP	%	IRC	%	FTP	%	SMTP	%
Port (successful)	93,429K	68.14	75,876	0.06	151,700	0.11	1,447K	1.06
Signature	94,326K	68.79	73,962	0.05	125,296	0.09	1,416K	1.03
on expected port	92,228K	67.3	71,467	0.05	98,017	0.07	1,415K	1.03
on other port	2,126K	1.6	2,495	0.00	27,279	0.02	265	0.00

Table 5.2: Comparison of signature-based detection vs. port-based detection (# connections).

Port	HTTP	IRC	FTP	SMTP	Other	No sig.
80	92.2M	59	0	0	41.1K	1.2M
6665-6669	1.2K	71.7K	0	0	4.2	524
21	0	0	98.0K	2	2.3K	52.5K
25	459	2	749	1.4M	195	31.9K

Table 5.3: Signature-based detection vs. port-based detection for well-known ports (# connections).

While inspecting the results we noticed that a connection sometimes triggers more than one signature. More detailed analysis reveals that l7-filter contains some signatures that are too general. For example, the signature for the Finger protocol matches simply if the first two characters at the beginning of the connection are printable characters. Such a signature will be triggered by a huge number of connections not using Finger. Another example comes from the "whois" signature. Accordingly, the data in Table 5.3 ignores matches by these two signatures.

Overall, the results show that the problem we pose does indeed already manifest operationally. Furthermore, because security analysis entails an adversary, what matters most is not the proportion of benign connections using ports other than those we might expect, but the prevalence of malicious connections doing so. We later discuss a number of such instances found operationally.

5.2.3 Existing NIDS Capabilities

Today's spectrum of intrusion detection and prevention systems offer powerful ways for detecting myriad forms of abuse. The simpler systems rely on searching for byte patterns within a packet stream, while the more complex perform extensive, stateful protocol analysis. In addition, some systems offer anomaly-based detection, comparing statistical characteristics of the monitored traffic against "normal network behavior," and/or specification-based detection, testing the characteristics against explicit specifications of allowed behavior.

For analyzing application-layer protocols, all systems of which we are aware depend upon port numbers.[2] While some can use signatures to *detect* other application-layer

[2] `DSniff` [DSn] is a network sniffer that extracts protocol-specific usernames and passwords independent

protocols, all only perform detailed protocol analysis for traffic identified via specific ports. Commercial systems rarely make details about their implementation available, and thus we must guess to what depth they analyze traffic. However, we have not seen an indication yet that any of them initiates stateful protocol analysis based on other properties than specific ports.

The most widely deployed open source NIDS, Snort [Roe99], does not per se ship with signatures for detecting protocols. However the Snort user community constantly contributes new open signatures [Bld], including ones for detecting IRC and FTP connections. Traditionally, Snort signatures are raw byte patterns. Newer versions of Snort also support regular expressions. As outlined in Chapter 2, the NIDS Bro ships with a *backdoor* [ZP00a] analyzer. Besides detecting interactive traffic by examining inter-packet intervals and packet size distributions it scans for some well-known protocols the analyzed payload for hard-coded byte patterns. In addition, Bro's signature matching engine [SP03b] is capable of matching the reassembled data stream of a connection against regular expression byte-patterns and leveraging the rich state of Bro's protocol decoders in the process. The commercial IntruShield system by Network Associates is primarily signature-based and ships with signatures for application detection, including SSH and popular P2P protocols. The technical details and the signatures do not appear accessible to the user. Therefore, it is unclear which property of a packet/stream triggers which signature or protocol violation. We also have some experience with Enterasys' Dragon system. It ships with a few signatures to match protocols such as IRC, but these do not appear to then enable full protocol analysis.

5.2.4 NIDS Limitations

It is useful to distinguish between the capability of detecting that a given application protocol is in use, versus then being able to continue to analyze that instance of use. Merely detecting the use of a given protocol can already provide actionable information; it might constitute a policy violation at a site for which a NIDS could institute blocking without needing to further analyze the connection. However, such a coarse-grained "go/no-go" capability has several drawbacks:

1. In some environments, such a policy may prove too restrictive or impractical due to the sheer size and diversity of the site. As user populations grow, the likelihood of users wishing to run legitimate servers on alternate ports rises.

2. Neither approach to application detection (byte patterns or statistical tests) is completely accurate (see Section 5.2.2). Blocking false-positives hinders legitimate operations, while failing to block false-negatives hinders protection.

3. Protocols that use non-fixed ports (e.g., Gnutella) can only be denied or allowed. Some of these, however, have legitimate applications as well as applications in violation of policy. For example, BitTorrent [BT] might be used for distributing

of ports. Its approach is similar to ours in that it uses a set of patterns to recognize protocols. It is however not a NIDS and does not provide any further payload analysis.

open-source software. Or, while a site might allow the use of IRC, including on non-standard ports, it highly desires to analyze all such uses in order to detect botnets.

In addition, some protocols are fundamentally difficult to detect with signatures, for example unstructured protocols such as Telnet. For Telnet, virtually any byte pattern at the beginning is potentially legitimate. Telnet can only be detected heuristically, by looking for plausible login dialogs [Pax99]. Another example is DNS, a binary protocol with no protocol identifier in the packet. The DNS header consists of 16-bit integers and bit fields which can take nearly arbitrary values. Thus, reliably detecting DNS requires checking the consistency across many fields. Similar problem exist for other binary protocols.

Another difficulty is that if an attacker knows the signatures, they may try to avoid the byte patterns that trigger the signature match. This means one needs "tight" signatures which comprehensively capture any use of a protocol for which an attacked end-system might engage. Finding such "tight" signatures can be particularly difficult due to the variety of end-system implementations and their idiosyncrasies.

5.3 Architecture

In this section we develop a framework for performing dynamic application-layer protocol analysis. Instead of a static determination of what analysis to perform based on port numbers, we introduce a processing path that dynamically adds and removes analysis components. The scheme uses a protocol detection mechanism as a trigger to activate analyzers (which are then given the entire traffic stream to date, including the portion already scanned by the detector), but these analyzers can subsequently decline to process the connection if they determine the trigger was in error. Currently, our implementation relies primarily on signatures for protocol detection, but our design allows for arbitrary other heuristics.

We present the design of the architecture in Section 5.3.1 and a realization of the architecture for the open-source NIDS Bro in Section 5.3.2. We finish with a discussion of the tradeoffs that arise in Section 5.3.3.

5.3.1 Design

Our design aims to achieve flexibility and power-of-expression, yet to remain sufficiently efficient for operational use. We pose the following requirements as necessary for these goals:

- **Detection scheme independence:**
  The architecture must accommodate different approaches to protocol detection Section 5.2.1. In addition, we should retain the possibility of using multiple techniques in parallel (e.g., complementing port-based detection with signature-based detection).

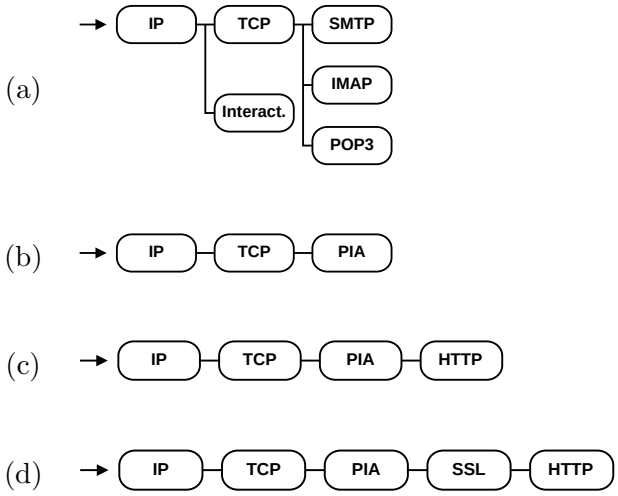


Figure 5.1: Example analyzer trees.

- **Dynamic analysis:**
  We need the capability of dynamically enabling or disabling protocol-specific analysis at any time during the lifetime of a connection. This goal arises because some protocol detection schemes cannot make a decision upon just the first packet of a connection. Once they make a decision, we must trigger the appropriate protocol analysis. Also, if the protocol analysis detects a false positive, we must have the ability to stop the analysis.

- **Modularity:**
  Reusable components allow for code reuse and ease extensions. This becomes particularly important for dealing with multiple network substacks (e.g., IP-within-IP tunnels) and performing in parallel multiple forms of protocol analysis (e.g., decoding in parallel with computing packet-size distributions).

- **Efficiency:**
  The additional processing required by the extended NIDS capabilities must remain commensurate with maintaining performance levels necessary for processing high-volume traffic streams.

- **Customizability:**
  The combination of analysis to perform needs to be easily adapted to the needs of the local security policy. In addition, the tradeoffs within the analysis components require configuration according to the environment.

To address these requirements we switch from the traditional static data analysis path to a dynamic one inside the NIDS's core. Traditional port-based NIDSs decide at the

time when they receive the first packet of each connection which analyses to perform. For example, given a TCP SYN packet with destination port 80, the NIDS will usually perform IP, TCP, and HTTP analysis for all subsequent packets of this flow. Our approach, on the other hand, relies on a per-connection data structure for representing the *data path*, which tracks what the system learns regarding what analysis to perform for the flow. If, for example, the payload of a packet on port 80/tcp – initially analyzed as HTTP – looks like an IRC session instead, we replace the HTTP analysis with IRC analysis.

We provide this flexibility by associating a tree structure with each connection. This tree represents the data path through various analysis components for all information transmitted on that connection (e.g., Figure 5.1(a)). Each node in this tree represents a self-contained unit of analysis, an *analyzer*. Each analyzer performs some kind of analysis on data received via an *input channel*, subsequently providing data via an *output channel*. The input channel of each node connects to an output channel of its data supplier (its predecessor in the data path tree). The input channel of the tree's root receives packets belonging to the connection/flow. Each intermediate node receives data via its input channel and computes analysis results, passing the possibly-transformed data to the next analyzer via its output channel.

Figure 5.1(a) shows an example of a possible analyzer tree for decoding email protocols. In this example, all analyzers (except `INTERACTIVE`) are responsible for the decoding of their respective network protocols. The packets of the connection first pass through the IP analyzer, then through the TCP analyzer. The output channel of the latter passes in replica to three analyzers for popular email protocols: SMTP, IMAP, and POP3. (Our architecture might instantiate such a tree for example if a signature match indicates that the payload looks like email but does not distinguish the application-layer protocol.) Note, though, that the analyzers need not correspond to a protocol, e.g., `INTERACTIVE` here, which examines inter-packet time intervals to detect surreptitious interactive traffic [ZP00a], performing its analysis in parallel to, and independent from, the TCP and email protocol analyzers.

To enable *dynamic* analysis, including analysis based on application-layer protocol identification, the analyzer tree changes over time. Initially, the analyzer tree of a new connection only contains those analyzers definitely needed. For example, if a flow's first packet uses TCP for transport, the tree will consist of an IP analyzer followed by a TCP analyzer.

We delegate application-layer protocol identification to a *protocol identification analyzer* (`PIA`), which works by applying a set of protocol detection heuristics to the data it receives. We insert this analyzer into the tree as a leaf-node after the TCP or UDP analyzer (see Figure 5.1(b)). Once the `PIA` detects a match for a protocol, it instantiates a child analyzer to which it then forwards the data it receives (see Figure 5.1(c)). However, the `PIA` also continues applying its heuristics, and if it finds another match it instantiates additional, or alternative, analyzers.

The analyzer tree can be dynamically adjusted throughout the entire lifetime of a connection by inserting or removing analyzers. Each analyzer has the ability to insert or remove other analyzers on its input and/or output channel. Accordingly, the tree changes

over time. Initially the PIA inserts analyzers as it finds matching protocols. Subsequently one of the analyzers may decide that it requires support provided by a missing analyzer and instantiates it; for instance, an IRC analyzer that learns that the connection has a compressed payload can insert a decompression analyzer as its predecessor.

If an analyzer provides data via its output channel, selecting successors becomes more complicated, as not all analyzers (including the TCP analyzer) have the capability to determine the protocol to which their output data conforms. In this case the analyzer can choose to instantiate *another* PIA and delegate to it the task of further inspecting the data. Otherwise it can simply instantiate the appropriate analyzer; see Figure 5.1(d) for the example of a connection using HTTP over SSL.

Finally, if an analyzer determines that it cannot cope with the data it receives over its input channel (e.g., because the data does not conform to its protocol), it removes its subtree from the tree.

This analyzer-tree design poses a number of technical challenges, ranging from the semantics of "input channels", to specifics of protocol analyzers, to performance issues. We now address each in turn.

First, the semantics of "input channels" differ across the network stack layers: some analyzers examine packets (e.g., IP, TCP, and protocols using UDP for transport), while others require byte-streams (e.g., protocols using TCP for transport). As the PIA can be inserted into arbitrary locations in the tree, it must cope with both types. To do so, we provide two separate input channels for each analyzer, one for packet input and one for stream input. Each analyzer implements the channel(s) suitable for its semantics. For example, the TCP analyzer accepts packet input and reassembles it into a payload stream, which serves as input to subsequent stream-based analyzers. An RPC analyzer accepts both packet and stream input, since RPC traffic can arrive over both UDP packets and TCP byte streams.

Another problem is the difficulty – or impossibility – of starting a protocol analyzer in the middle of a connection. For example, an HTTP analyzer cannot determine the correct HTTP state for such a partial connection. However, most non-port-based protocol detection schemes can rarely identify the appropriate analyzer(s) upon inspecting just the first packet of a connection. Therefore it is important that the PIA buffers the beginning of each input stream, up to a configurable threshold (default 4KB in our implementation). If the PIA decides to insert a child analyzer, it first forwards the data in the buffer to it before forwarding new data. This gives the child analyzer a chance to receive the total payload if detection occurred within the time provided by the buffer. If instantiation occurs only after the buffer has overflowed, the PIA only instantiates analyzers capable of resynchronizing to the data stream, i.e., those with support for partial connections.

Finally, for efficiency the PIA requires very lightweight execution, as we instantiate at least one for every flow/connection. To avoid unnecessary resource consumption, our design factors out the user configuration, tree manipulation interface, and functions requiring permanent state (especially state independent of a connection's lifetime) into a single central management component which also instantiates the initial analyzer trees.

In summary, the approach of generalizing the processing path to an analyzer tree provides numerous new possibilities while addressing the requirements. We can: *(i)* readily plug in new protocol detection schemes via the `PIA`; *(ii)* dynamically enable and disable analyzers at any time (protocol semantics permitting); *(iii)* enable the user to customize and control the processing via an interface to the central manager; *(iv)* keep minimal the overhead of passing data along the tree branches; *(v)* support pure port-based analysis using a static analyzer tree installed at connection initialization; and *(vi)* support modularity by incorporating self-contained analyzers using a standardized API, which allows any protocol analyzer to also serve as a protocol *verifier*.

5.3.2 Implementation

We implemented our design within the *Bro* NIDS, leveraging both its already existing set of protocol decoders and its signature-matching engine (see Chapter 2). Remember that, like other systems, Bro performs comprehensive protocol-level analysis using a static data path, relying on port numbers to identify application-layer protocols. However, its modular design encourages application-layer decoders to be mainly self-contained, making it feasible to introduce a dynamic analyzer structure as discussed in Section 5.3.1.

We implemented the `PIA`, the analyzer trees, and the central manager, terming this modification of Bro as `PIA-Bro`; for details see [Mai05]. We use signatures as our primary protocol-detection heuristic (though see below), equipping the `PIA` with an interface to Bro's signature-matching engine such that analyzers can add signatures corresponding to their particular protocols. For efficiency, we restricted the signature matching to the data buffered by the `PIA`s; previous work[SSW04, MP05] indicates that for protocol detection it suffices to examine at most a few KB at the beginning of a connection. By skipping the tails, we can avoid performing pattern matching on the bulk of the total volume, exploiting the heavy-tailed nature of network traffic [PF95].

In addition to matching signatures, our implementation can incorporate other schemes for determining the right analyzers to activate. First, the `PIA` can still activate analyzers based on a user-configured list of well-known ports.[3] In addition, each protocol analyzer can register a specific detection function. The `PIA` then calls this function for any new data chunk, allowing the use of arbitrary heuristics to recognize the protocol. Finally, leveraging the fact that the central manager can store state, we also implemented a *prediction table* for storing anticipated future connections along with a corresponding analyzer. When the system eventually encounters one of these connections, it inserts the designated analyzer into the tree. (See Section 5.4.2 below for using this mechanism to inspect FTP data-transfer connections.) Together these mechanisms provide the necessary flexibility for the connections requiring dynamic detection, as well as good performance for the bulk of statically predictable connections.

As Bro is a large and internally quite complex system, we incrementally migrate its protocol analyzers to use the new framework. Our design supports this by allowing old-style and new-style data paths to coexist: for those applications we have adapted, we

[3] This differs from the traditional Bro, where the set of well-known ports is hard-coded.

gain the full power of the new architecture, while the other applications remain usable in the traditional (static ports) way.

For our initial transition of the analyzers we have concentrated on protocols running on top of TCP. The Bro system already encapsulates its protocol decoders into separate units; we redesigned the API of these units to accommodate the dynamic analyzer structure. We have converted four of the system's existing application-layer protocol decoders to the new API: FTP, HTTP, IRC, and SMTP.[4] The focus on TCP causes the initial analyzer tree to always contain the IP and TCP analyzers. Therefore we can leverage the existing static code and did not yet have to adapt the IP and TCP logic to the new analyzer API. We have, however, already moved the TCP stream reassembly code into a separate "Stream" analyzer. When we integrate UDP into the framework, we will also adapt the IP and TCP components.

The Stream analyzer is one instance of a *support analyzer* which does not directly correspond to any specific protocol. Other support analyzers provide functionality such as splitting the payload-stream of text-based protocols into lines, or expanding compressed data.[5] We have not yet experimented with pipelining protocol analyzers such as those required for tunnel decapsulation, but intend to adapt Bro's SSL decoder next to enable us to analyze HTTPS and IMAPS in a pipelined fashion when we provide the system with the corresponding secret key.

5.3.3 Tradeoffs

Using the `PIA` architecture raises some important tradeoffs to consider since protocol recognition/verification is now a multi-step process. First, the user must decide what kinds of signatures to apply to detect *potential* application-layer protocols. Second, if a signature matches it activates the appropriate protocol-specific analyzer, at which point the system must cope with possible false positives; when and how does the analyzer fail in this case? Finally, we must consider how an attacker can exploit these tradeoffs to subvert the analysis.

The first tradeoff involves choosing appropriate signatures for the protocol detection. On the one hand, the multi-step approach allows us to *loosen* the signatures that initially detect protocol candidates. Signatures are typically prone to false alerts, and thus when used to generate alerts need to be specified as tight as possible – which in turn very often leads to false negatives, i.e., undetected protocols in this context. However, by relying on analyzers *verifying* protocol conformance after a signature match, false positives become more affordable. On the other hand, signatures should not be *too* lose: having an analyzer inspect a connection is more expensive than performing pure pattern matching. In addition, we want to avoid enabling an attacker to trigger expensive protocol processing by deliberately crafting bogus connection payloads.

Towards these ends, our implementation uses bidirectional signatures [SP03b], which

[4] Note that it does not require much effort to convert an existing application-layer analyzer to the new API. For example, the SMTP analyzer took us about an hour to adapt.

[5] Internally, these support analyzers are implemented via a slightly different interface, see [Mai05] for details.

```
signature http_server {                    # Server-side signature
  ip-proto == tcp                          # Examine TCP packets.
  payload /^HTTP\/[0-9]/                   # Look for server response.
  tcp-state responder                      # Match responder-side of connection.
  requires-reverse-signature http_client # Require client-side signature too.
  enable "http"                            # Enable analyzer upon match.
}
signature http_client {                    # Client-side signature
  ip-proto == tcp                          # Examine TCP packets.
  payload /^[[:space:]]*GET[[:space:]]*/ # Look for requests [simplified]
  tcp-state originator                     # Match originator-side of connection.
}
```

Figure 5.2: Bidirectional signature for HTTP.

only match if *both* endpoints of a connection appear to participate in the protocol. If an attacker only controls one side (if they control both, we are sunk in many different ways), they thus cannot force activation of protocol analyzers by themselves. In practice, we in fact go a step further: before assuming that a connection uses a certain protocol, the corresponding analyzer must also *parse* something meaningful for both directions. This significantly reduces the impact of false positives. Figure 5.2 shows an example of the signature we currently use for activating the HTTP analyzer. (We note that the point here is not about signature quality; for our system, signatures are just one part of the NIDS's configuration to be adapted to the user's requirements.)

Another tradeoff to address is *when* to decide that a connection uses a certain protocol. This is important if the use of a certain application violates a site's security policy and should cause the NIDS to raise an alert. A signature-match triggers the activation of an analyzer that analyzes and verifies the protocol usage. Therefore, before alerting, the system waits until it sees that the analyzer is capable of handling the connection's payload. In principle, it can only confirm this with certainty once the connection completes. In practice, doing so will delay alerts significantly for long-term connections. Therefore our implementation assumes that if the analyzer can parse the connection's *beginning*, the rest of the payload will also adhere to the same protocol. That is, our system reports use of a protocol if the corresponding analyzer is (still) active after the exchange of a given volume of payload, or a given amount of time passes (both thresholds are configurable).

Another tradeoff stems from the question of protocol verification: at what point should an analyzer indicate that it *cannot* cope with the payload of a given connection? Two extreme answers: (i) reject immediately when something occurs not in accordance with the protocol's definition, or (ii) continue parsing come whatever may, in the hope that eventually the analyzer can resynchronize with the data stream. Neither extreme works well: real-world network traffic often stretches the bounds of a protocol's specification, but trying to parse the entire stream contradicts the goal of verifying the protocol. The right balance between these extremes needs to be decided on a per-protocol basis. So

far, we have chosen to reject connections if they violate basic protocol properties. For example, the FTP analyzer complains if it does not find a numeric reply-code in the server's response. However, we anticipate needing to refine this decision process for instances where the distinction between clear noncompliance versus perhaps-just-weird behavior is less crisp.

Finally, an attacker might exploit the specifics of a particular analyzer, avoiding detection by crafting traffic in a manner that the analyzer believes reflects a protocol violation, while the connection's other endpoint still accepts or benignly ignores the data. This problem appears fundamental to protocol detection, and indeed is an instance of the more general problem of evasion-by-ambiguity [PTN98, HKP01], and, for signatures, the vulnerability of NIDS signatures to attack variants. To mitigate this problem, we inserted indirection into the decision process: in our implementation, an analyzer *never* disables itself, even if it fails to parse its inputs. Instead, upon severe protocol violations it generates Bro events that a user-level policy script then acts upon. The default script is fully customizable, capable of extension to implementing arbitrary complex policies such as disabling the analyzer only after repeated violations. This approach fits with the Kerkhoff-like principle used by the Bro system: the code is open, yet sites code their specific policies in user-level scripts which they strive to keep secret.

5.4 Applications

We now illustrate the increased detection capabilities that stem from realizing the `PIA` architecture within Bro, using three powerful example applications: (i) reliable detection of applications running on non-standard ports, (ii) payload inspection of FTP data transfers, and (iii) detection of IRC-based bot clients and servers. All three schemes run in day-to-day operations at UCB, MWN, and LBNL (see Section 2.3), where they have already identified a large number of compromised hosts which the sites' traditional security monitoring could not directly detect.

5.4.1 Detecting Uses of Non-standard Ports

As pointed out earlier, a `PIA` architecture gives us the powerful ability to verify protocol usage and extract higher-level semantics. To take advantage of this capability, we extended the reporting of `PIA-Bro`'s analyzers. Once the NIDS knows which protocol a connection uses, it can leverage this to extract more semantic context. For example, HTTP is used by a wide range of other protocols as a transport protocol. Therefore, an alert such as "connection uses HTTP on a non-standard port 21012", while useful, does not tell the whole story; we would like to know *what* that connection then does. We extended `PIA-Bro`'s HTTP analysis to distinguish the various protocols using HTTP for transport by analyzing the HTTP dialog. Kazaa, for example, includes custom headers lines that start with `X-Kazaa`. Thus, when this string is present, the NIDS generates a message such as "connection uses Kazaa on port 21021". We added patterns for detecting Kazaa, Gnutella, BitTorrent, Squid, and SOAP applications running over HTTP. In

addition, the HTTP analyzer extracts the "Server" header from the HTTP responses, giving an additional indication of the underlying application.

We currently run the dynamic protocol detection for FTP, HTTP, IRC, and SMTP on the border routers of all three environments, though here we primarily report on experiences at UCB and MWN. As we have particular interest in the use of non-standard ports, and to reduce the load on `PIA-Bro`, we exclude traffic on the analyzers' well-known ports from our analysis. For "well-known" we consider those for which a traditional Bro triggers application-layer analysis. These are port 21 for FTP, ports 6667/6668 for IRC, 80/81/631/1080/3128/8000/8080/8888 for HTTP (631 is IPP), and 25 for SMTP. We furthermore added 6665/6666/6668/6669/7000 for IRC, and 587 for SMTP as we encountered them on a regular basis. To further reduce the load on the monitor machines, we excluded a few high volume hosts, including the PlanetLab servers at UCB and the heavily accessed `leo.org` domain at MWN. Note, that this setup prevents `PIA-Bro` from finding some forms of port abuse, e.g., an IRC connection running on the HTTP port. We postpone this issue to Section 5.5.

At both UCB and MWN, our system quickly identified many servers which had gone unnoticed. We consider an IP address to run a server, if it accepts connections and participates in the protocol exchange. Due to NAT address space, we may underestimate or overestimate the number of actual hosts. At UCB, the system found within a day 6 internal and 17 remote FTP servers, 568/54830 HTTP servers (!), 2/33 IRC servers, and 8/8 SMTP servers running on non-standard ports. At MWN, during a similar period, we found 3/40 FTP, 108/18844 HTTP, 3/58 IRC, and 3/5 SMTP servers.

For FTP, IRC, and SMTP we manually checked whether the internal hosts were indeed running the detected protocol; for HTTP, we verified a subset. Among the checked servers we found only one false positive: `PIA-Bro` incorrectly flagged one SMTP server due to our choice regarding how to cope with false positives: as discussed in Section 5.3.3, we choose to not wait until the end of the connection before alerting. In this case, the SMTP analyzer correctly reported a protocol violation for the connection, but it did so only *after* our chosen maximal interval of 30 seconds had already passed; the server's response took quite some time. In terms of connections, HTTP is, not surprisingly, the most prevalent of the four protocols: at UCB during the one-day period, 99% of the roughly 970,000 reported off-port connections are HTTP. Of these, 28% are attributed to Gnutella, 22% to Apache, and 12% to Freechal [FC]. At MWN, 92% of the 250,000 reported connections are HTTP, and 7% FTP (of these 70% were initiated by the same host). Of the HTTP connections, roughly 21% are attributed to BitTorrent, 20% to Gnutella, and 14% to SOAP.

That protocol analyzers can now extract protocol semantics not just for HTTP but also for the other protocols proves to be quite valuable. `PIA-Bro` generates detailed protocol-level log files for all connections. A short glance at, for example, an FTP log file quickly reveals whether an FTP server deserves closer attention. Figure 5.3 shows an excerpt of such a log file for an obviously compromised host at MWN. During a two-week period, we found such hosts in both environments, although UCB as well as MWN already deploy Snort signatures supposed to detect such FTP servers.

With `PIA-Bro`, any protocol-level analysis automatically extends to non-standard

```
xxx.xxx.xxx.xxx/2373 > xxx.xxx.xxx.xxx/5560 start
response (220 Rooted Moron Version 1.00 4 WinSock ready...)
USER ops (logged in)
SYST (215 UNIX Type: L8)
[...]
LIST -al (complete)
TYPE I (ok)
SIZE stargate.atl.s02e18.hdtv.xvid-tvd.avi (unavail)
PORT xxx,xxx,xxx,xxx,xxx,xxx (ok)
*STOR stargate.atl.s02e18.hdtv.xvid-tvd.avi, NOOP (ok)
ftp-data video/x-msvideo 'RIFF (little-endian) data, AVI'
[...]
response (226 Transfer complete.)
[...]
QUIT (closed)
```

Figure 5.3: Application-layer log of an FTP-session to a compromised server (anonymized / edited for clarity).

ports. For example, we devised a detector for HTTP proxies which matches HTTP requests going into a host with those issued by the same system to external systems. With the traditional setup, it can only report proxies on well-known ports; with `PIA-Bro` in place, it has correctly identified proxies inside the UCB and MWN networks running on different ports;[6] two of them were world-open.

It depends on a site's policy whether offering a service on a non-standard port constitutes a problem. Both university environments favor open policies, generally tolerating offering non-standard services. For the internal servers we identified, we verified that they meet at least basic security requirements. For all SMTP servers, for example, we ensured that they do not allow arbitrary relaying. One at MWN which did was quickly closed after we reported it, as were the open HTTP proxies.

5.4.2 Payload Inspection of FTP Data

According to the experience of network operators, attackers often install FTP servers on non-standard ports on machines that they have compromised. `PIA-Bro` now not only gives us a reliable way to detect such servers but, in addition, can *examine* the transferred files. This is an impossible task for traditional NIDSs, as FTP is a protocol for which for the data-transfer connections by design use arbitrary port combinations. For security monitoring, inspecting the *transferred* data for files exchanged via non-standard-port services enables alerts on sensitive files such as system database accesses or download/upload of virus-infected files. We introduced a new *file analyzer* to perform such analysis for FTP data connections, as well as for other protocols used to transfer files. When `PIA-Bro` learns, e.g., via its analysis of the control session, of an upcoming

[6] As observing both internal as well as outgoing requests at *border* is rather unusual, this detection methodology generally detects proxies other than the site's intended ones.

data transfer, it adds the expected connection to the dynamic prediction table (see Section 5.3.2). Once this connection is seen, the system instantiates a file analyzer, which examines the connection's payload.

The file analyzer receives the file's full content as a reassembled stream and can utilize any file-based intrusion detection scheme. To demonstrate this capability, our file-type identification for `PIA-Bro` leverages `libmagic` [Mag], which ships with a large library of file-type characteristics. This allows `PIA-Bro` to log the file-type's textual description as well as its MIME-type as determined by `libmagic` based on the payload at the beginning of the connection. Our extended FTP analyzer logs – and potentially alerts on – the file's content type. Figure 5.3 shows the result of the file type identification in the `ftp-data` line. The NIDS categorizes the data transfer as being of MIME type `video/x-msvideo` and, more specifically, as an AVI movie. As there usually are only a relatively small number of ftp-data connections, this mechanism imposes quite minimal performance overhead.

We envision several extensions to the file analyzer. One straight-forward improvement, suggested to us by the operators at LBNL, is to match a file's name with its actual content (e.g., a file `picture.gif` requested from a FTP server can turn out to be an executable). Another easy extension is the addition of an interface to a virus checker (e.g., Clam AntiVirus [Cla]). We also plan to adapt other protocol analyzers to take advantage of the file analyzer, such as TFTP (once `PIA-Bro` has support for UDP) and SMTP. TFTP has been used in the past by worms to download malicious code [Bla]. Similarly, SMTP can pass attachments to the file analyzer for inspection. SMTP differs from FTP in that it transfers files in-band, i.e., inside the SMTP session, rather than out-of-band over a separate data connection. Therefore, for SMTP there is no need to use the dynamic prediction table. Yet, we need the capabilities of `PIA-Bro` to pipeline the analyzers: first the SMTP analyzer strips the attachments' MIME-encoding, then the file analyzer inspects the file's content.

5.4.3 Detecting IRC-based Botnets

Attackers systematically install trojans together with *bots* for remote command execution on vulnerable systems. Together, these form large *botnets* controlled by a human *master* that communicates with the bots by sending commands. Such commands can be to flood a victim, send spam, or sniff confidential information such as passwords. Often, thousands of individual bots are controlled by a single master [Hon05], constituting one of the largest security threats in today's Internet.

The IRC protocol [Kal00] is a popular means for communication within botnets as it has some appealing properties for remote control: it provides *public channels* for one-to-many communication, with *channel topics* well-suited for holding commands; and it provides *private channels* for one-to-one communication.

It is difficult for a traditional NIDS to reliably detect members of IRC-based botnets. Often, the bots never connect to a standard IRC server – if they did they would be easy to track down – but to a separate *bot-server* on some non-IRC port somewhere in the Internet. However, users also sometimes connect to IRC servers running on non-standard

```
Detected bot-servers:
IP1 - ports 9009,6556,5552 password(s) <none> last 18:01:56
 channel #vec:
 topic ".asc pnp 30 5 999 -b -s|.wksescan 10 5 999 -b -s|[...]"
 channel #hv:
 topic ".update http://XXX/image1.pif f'', password(s) XXX"
[...]
Detected bots:
IP2 - server IP3 usr 2K-8006 nick [P00|DEU|59228] last 14:21:59
IP4 - server IP5 usr XP-3883 nick [P00|DEU|88820] last 19:28:12
[...]
```

Figure 5.4: Excerpt of the set of detected IRC bots and bot-servers (anonymized / edited for clarity).

ports for legitimate (non-policy-violating) purposes. Even if a traditional NIDS has the capability of detecting IRC servers on non-standard ports, it lacks the ability to then distinguish between these two cases.

We used `PIA-Bro` to implement a reliable bot-detector that has already identified a significant number of bot-clients at MWN and UCB. The detector operates on top of the IRC analyzer and can thus perform protocol-aware analysis of *all* detected IRC sessions. To identify a bot connection, it uses three heuristics. First, it checks if the client's nickname matches a (customizable) set of regular expression patterns we have found to be used by some botnets (e.g., a typical botnet "nick" identifier is `[0]CHN|3436036`). Second, it examines the channel topics to see if it includes a typical botnet command (such as `.advscan`, which is used by variants of the SdBot family[Hon05]). Third, it flags new clients that establish an IRC connection to an already identified bot-server as bots. The last heuristic is very powerful, as it leverages the state that the detector accumulates over time and does not depend on any particular payload pattern. Figure 5.4 shows an excerpt of the list of known bots and bot-servers that one of our operational detectors maintains. This includes the server(s) contacted as well as the timestamp of the last alarming IRC command. (Such timestamps aid in identifying the owner of the system in NAT'd or DHCP environments.) For the servers, the list contains channel information, including topics and passwords, as well as the clients that have contacted them.

At MWN the bot-detector quickly flagged a large number of bots. So far, it has identified more than 100 distinct local addresses. To exclude the danger of false positives, we manually verified a subset. To date, we have not encountered any problems with our detection. Interestingly, at UCB there are either other kinds of bots, or not as many compromised machines; during a two-week time period we reported only 15 internal hosts to the network administrators. We note that the NIDS, due to only looking for patterns of *known* bots, certainly misses victims; this is the typical drawback of such a misuse-detection approach, but one we can improve over time as we learn more signatures through other means.

Of the detected bots at MWN, only five used static IP addresses, while the rest used

		Stock-Bro	PIA-Bro	PIA-Bro-M4K
Config-A	Standard	3335	3254	—
	Standard + sigs	3843	3778	—
Config-A'	Standard	3213	3142	—
Config-B	All TCP pkts	3584	3496	—
Config-C	All TCP pkts + sigs	4446	4436	3716
	All TCP pkts + sigs + reass.	—	4488	3795

Table 5.4: CPU user times on subset of the trace (secs; averaged over 3 runs each; standard deviation always < 13s).

IP addresses from a NAT'd address range, indicating that most of them are private, student-owned machines. It is very time-consuming for the MWN operators to track down NAT'd IP addresses to their actual source. Worse, the experience at MWN is that even if they do, many of the owners turn out to not have the skills to remove the bot. Yet, it is important that such boxes cannot access the Internet.

The MWN operators accomplish this with the help of our system. They installed a blocking system for MWN's NAT subnets to which we interface with our system. The operators have found the system's soundness sufficiently reliable for flagging bots, that they enabled it to block all reported bots *automatically*. They run this setup operationally, and so far without reporting to us any complaints. In the beginning, just after our system went online, the average number of blocked hosts increased by 10-20 addresses. After about two weeks of operation, the number of blocked hosts has almost settled back to the previous level, indicating that the system is effective: the number of bots has been significantly reduced.

Finally, we note that our detection scheme relies on the fact that a bot uses the IRC protocol in a fashion which conforms to the standard IRC specification. If the bot uses a custom protocol dialect, the IRC analyzer might not be able to parse the payload. This is a fundamental problem similar to the one we face if a bot uses a proprietary protocol. More generally we observe that the setting of seeing malicious clients *and* servers violates an important assumption of many network intrusion detection systems: an attacker does not control both endpoints of a connection [Pax99]. If he does, any analysis is at best a heuristic.

5.5 Evaluation

We finish with an assessment of the performance impact of the PIA architecture and a look at the efficacy of the multi-step protocol recognition/verification process. The evaluation confirms that our implementation of the PIA framework does not impose an undue performance overhead.

5.5.1 CPU Performance

To understand the performance impact of the PIA extensions to Bro, we conduct CPU measurements for both the unmodified Bro (developer version 1.1.52), referred to as Stock-Bro, and PIA-Bro running on the first 66 GB of the mwn-full-packets trace which corresponds to 29 minutes of traffic. (This trace again excludes the domain leo.org.) For the analysis we used one of our dual-Opteron systems with 2GB RAM, running FreeBSD 6.0.

In addition to the processing of the (default) Stock-Bro configuration, PIA-Bro must also perform four types of additional work: (i) examining *all* packets; (ii) performing *signature matches* for many packets; and (iii) buffering and reassembling the beginnings of all streams to enable reliable protocol detection; (iv) performing application-layer protocol analysis on additionally identified connections. In total, these constitute the cost we must pay for PIA-Bro's additional detection capabilities.

To measure the cost of each additional analysis element, we enable them one by one, as reported in Table 5.4. We begin with basic analysis (Config-A): Bro's generation of one-line connection summaries, as well as application-layer protocol analysis for FTP, HTTP, IRC, SMTP connections, as identified via port numbers. The first line reports CPU times for both versions of Bro performing this regular analysis, and the second line when we also enable Bro signatures corresponding to those used by PIA-Bro for protocol detection. We find the runtimes of the two systems quite similar, indicating that our implementation of the PIA architecture does not add overhead to Bro's existing processing. (The runtime of PIA-Bro is slightly less than Stock-Bro due to minor differences in their TCP bytestream reassembly process; this also leads PIA-Bro to make slightly fewer calls to Bro's signature engine for the results reported below. The runtimes of both systems exceed the duration of the trace, indicating that we use a configuration which, in this environment, requires multiple NIDS instances in live operation.)

With Config-A, the systems only need to process a subset of all packets: those using the well-known ports of the four protocols, plus any with TCP SYN/FIN/RST control packets (which Bro uses to generate generic TCP connection summaries). Bro uses a BPF [MJ93] filter to discard any other packets. However, PIA-Bro cannot use this filtering because by its nature it needs to examine *all* packets. This imposes a significant performance penalty, which we assess in two different ways.

First, we prefilter the trace to a version containing only those packets matched by Bro's BPF filter, which in this case results in a trace just under 60% the size of the original. Running on this trace rather than the original approximates the benefit Bro obtains when executing on systems that use in-kernel BPF filtering for which captured packets must be copied to user space but discarded packets do not require a copy. The Table shows these timings as Config-A'. We see that, for this environment and configuration, this cost for using PIA is minor, about 3.5%.

Second, we manually change the filters of both systems to include all TCP packets (Config-B). The user time increases by a fairly modest 7.5% for Stock-Bro and 7.4% for PIA-Bro compared to Config-B. Note that here we are not yet enabling PIA-Bro's additional functionality, but are only assessing the cost to Bro of processing the entire

packet stream using the above configuration; this entails little extra work for Bro since it does not perform application analysis on the additional packets.

PIA-Bro's use of signature matching also imposes overhead. While most major NIDSs rely on signature matching, the Bro system's default configuration does not. Accordingly, applying the PIA signatures to the packet stream adds to the system's load. To measure its cost, we added signature matching to the systems' configuration (second line of the table, as discussed above). The increase compared to Config-A is about 15–16%.

When we activate signatures for Config-B, we obtain Config-C, which now enables essentially the full PIA functionality. This increases runtime by 24–27% for Stock-Bro and PIA-Bro, respectively. Note that by the comparison with Stock-Bro running equivalent signatures, we see that capturing the entire packet stream and running signatures against it account for virtually all of the additional overhead PIA-Bro incurs.

As the quality of the signature matches improves when PIA-Bro has access to the reassembled payload of the connections, we further consider a configuration based on Config-C that also reassembles the data which the central manager buffers. This configuration only applies to PIA-Bro, for which it imposes a performance penalty of 1.2%. The penalty is so small because most packets arrive in order [DP05], and we only reassemble the first 4KB (the PIA buffer size).

As we can detect most protocols within the first KBs (see Section 5.3.2), we also evaluated a version of PIA-Bro that restricts signature matching to only the first 4KB. This optimization, which we annotate as PIA-Bro-M4K, yields a performance gain of 16.2%. Finally, adding reassembly has again only a small penalty (2.1%).

In summary, for this configuration we can obtain nearly the full power of the PIA architecture (examining all packets, reassembling and matching on the first 4KB) at a performance cost of about 13.8% compared to Stock-Bro. While this is noticeable, we argue that the additional detection power provided merits the expenditure. We also note that the largest performance impact stems from applying signature matching to a large number of packets, for which we could envision leveraging specialized hardware to speed up. Finally, because we perform dynamic protocol analysis on a per-connection basis, the approach lends itself well to front-end load-balancing.

5.5.2 Detection Performance

We finish with a look at the efficacy of the PIA architecture's multi-step analysis process. To do so, we ran PIA-Bro with all adapted analyzers (HTTP, FTP, IRC, SMTP) on the 24-hour mwn-full-packets trace, relying only on our bidirectional PIA-signatures for protocol detection, i.e., no port based identification. (Note that as these signatures differ from the L7-signatures used in Section 5.2, the results are not directly comparable.) PIA-Bro verifies the detection as discussed in Section 5.3.3, i.e., when the connection has either run for 30 seconds or transferred 4 KB of data (or terminated).

Our goal is to understand the quality of its detection in terms of false positives and false negatives. In trading off these two, we particularly wish to minimize false positives, as our experience related in Section 5.4 indicates that network operators strongly desire actionable information when reporting suspected bot hosts or surreptitious servers.

	Detected and verified non-std. port	Rejected by analyzer non std. port	Rejected by analyzer std. port
HTTP	1,283,132	21,153	146,202
FTP	14,488	180	1,792
IRC	1,421	91	3
SMTP	69	0	1,368

Table 5.5: # of connections with detection and verification.

Table 5.5 breaks down `PIA-Bro`'s detections as follows. The first column shows how often (i) a protocol detection signature flagged the given protocol as running on a non-standard port, for which (ii) the corresponding analyzer verified the detection. With strong likelihood, these detections reflect actionable information.

The second and third columns list how often the analyzer did *not* agree with the detection, but instead rejected the connection as exhibiting the given protocol, for non-standard and standard ports, respectively. Thus, the second column highlights the role the analyzer plays in reducing false positives; had we simply employed signatures without subsequent verification, then in these cases we would have derived erroneous information.

The third column, on the other hand, raises questions regarding to what degree our protocol detection might be missing instances of given protocols. While we have not yet systematically assessed these rejections, those we have manually inspected have generally revealed either a significant protocol failure, or indeed an application other than that associated with the standard port. Examples of the former include errors in HTTP headers, non-numeric status codes in FTP responses, mismatches in SMTP dialogs between requests and responses, use of SMTP reply codes beyond the assigned range, and extremely short or mismatched IRC replies.

While we detect a large number of verified connections on non-standard ports – with the huge number of HTTP connections primarily due to various P2P applications – for this trace the only instance we found of a different protocol running on a privileged standard port was a (benign) IRC connection running on 80/tcp. On the unprivileged ports used for IRC, however, we found a private Apache HTTP server, a number of video-on-demand servers, and three FTP servers used for (likely illicit) music-sharing. (Note that, different than in Section 5.2.2, when looking for protocols running on standard ports, we can only detect instances of FTP, HTTP, IRC, and SMTP; also, protocols running *on top* of HTTP on port 80 are not reported.)

Finally, Figure 5.5 and Figure 5.6 show the diversity of the non-standard ports used by different types of servers. The x-axes give the port number used and the y-axis the number of connections whose servers resided on that port (log-scaled). The 22,647 HTTP servers we detected used 4,024 different non-standard ports, some involving more than 100,000 connections (see Figure 5.5). We checked the top ten HTTP ports (which account for 88% of the connections) and found that most are due to a number of PlanetLab hosts (ports 312X, 212X), but also quite a large number are due to P2P applications, with

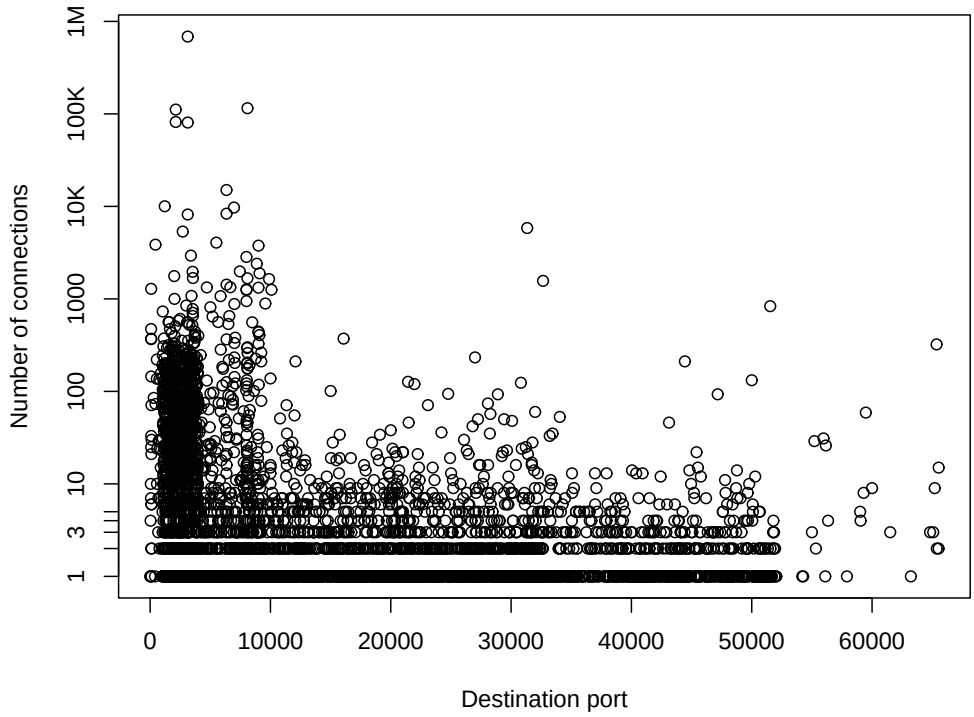


Figure 5.5: Connections using the HTTP protocol.

Gnutella (port 6346) contributing the largest number of distinct servers. Similar observations, but in smaller numbers, hold for IRC, FTP, and SMTP, for which we observed 60, 81, and 11 different non-standard server ports, respectively (see Figure 5.6). These variations, together with the security violations we discussed in Section 5.4, highlight the need for dynamic protocol detection.

5.6 Conclusion

In this chapter we have developed a general NIDS framework which overcomes the limitations of traditional, port-based protocol analysis. The need for this capability arises because in today's networks an increasing share of the traffic resists correct classification using TCP/UDP port numbers. For a NIDS, such traffic is particularly interesting, as a common reason to avoid well-known ports is to evade security monitoring and policy enforcement. Still, today's NIDSs rely exclusively on ports to decide which higher-level protocol analysis to perform.

Our framework introduces a dynamic processing path that adds and removes analysis components as required. The scheme uses protocol detection mechanisms as triggers to activate analyzers, which can subsequently decline to process the connection if they determine the trigger was in error. The design of the framework is independent of any

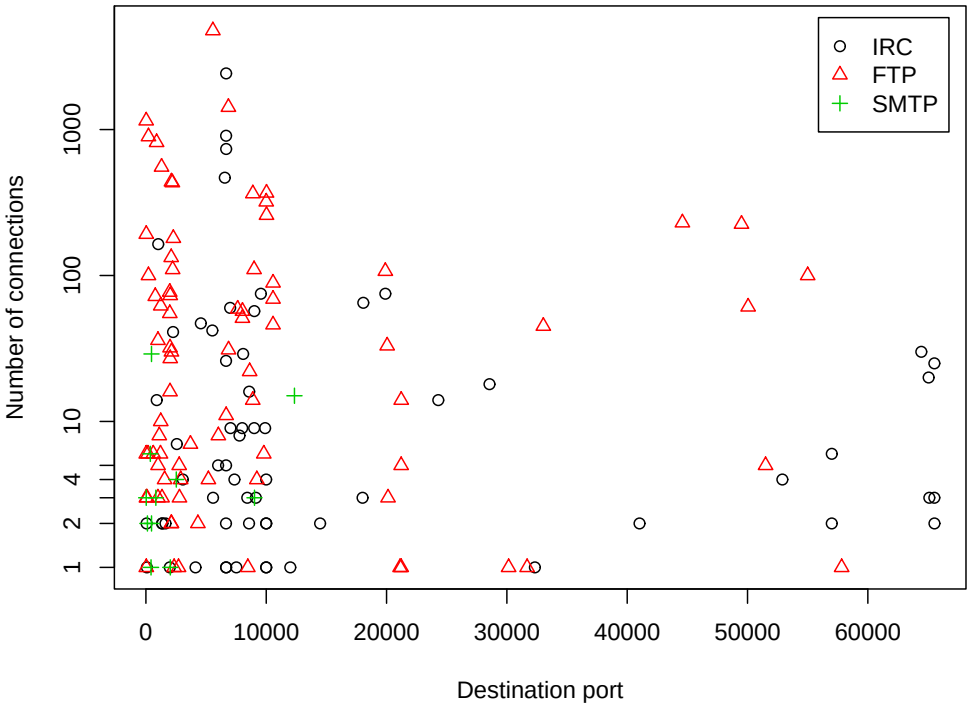


Figure 5.6: Connections using the IRC, FTP, SMTP protocol .

particular detection scheme and allows for the addition/removal of analyzers at arbitrary times. The design provides a high degree of modularity, which allows analyzers to work in parallel (e.g., to perform independent analyses of the same data), or in series (e.g., to decapsulate tunnels).

We implemented our design within the open-source Bro NIDS. We adapted several of the system's key components to leverage the new framework, including the protocol analyzers for HTTP, IRC, FTP, and SMTP, as well as leveraging Bro's signature engine as an efficient means for performing the initial protocol detection that is then verified by Bro's analyzers.

Prototypes of our extended Bro system currently run at the borders of three large-scale operational networks. Our example applications – reliable recognition of uses of non-standard ports, payload inspection of FTP data transfers, and detection of IRC-based botnet clients and servers – have already exposed a significant number of security incidents at these sites. Due to its success, the MWN site has integrated our bot-detection into dynamic blocking of production traffic.

In the near future, we will migrate the remainder of Bro's analyzers to the new framework. From our experiences to date, it appears clear that using dynamic protocol analysis operationally will significantly increase the number of security breaches we can detect.

6 Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic

6.1 Introduction

As we have seen in the last chapters, it is, due to resource constraints, infeasible for NIDS to perform all possible analyses. As a consequence, the alerts the systems output contain very limited context of the attack. The best source for the all additional context are network packet traces with full packet contents. These can also prove invaluable for trouble shooting network problems or problems with the detection process of a NIDS. Yet in many operational environments the sheer volume of the traffic makes it infeasible to capture the entire stream or retain even significant subsets for extended amounts of time. Of course, for both troubleshooting and security forensics, only a very small proportion of the traffic actually turns out to be pertinent. The problem is that one has to decide *beforehand*, when configuring a traffic monitor, what context will turn out to be relevant *retrospectively* to investigate incidents.

Only in low volume environments can one routinely bulk-record all network traffic using tools such as `tcpdump` [TCP]. Rising volumes inevitably require filtering. For example, at the Lawrence Berkeley National Laboratory (*LBNL*), a medium-size Gbps environment, the network traffic averages 1.5 TB/day, right at the edge of what can be recorded using commodity hardware. The site has found it vital to record traffic for analyzing possible security events, but cannot retain the full volume. Instead, the operators resort to a `tcpdump` filter with *85 terms* describing the traffic to skip – omitting any recording of key services such as HTTP, FTP data, X11 and NFS, as well as skipping a number of specific high-volume hosts, and all non-TCP traffic. This filter reduces the volume of recorded traffic to about 4% of the total.

At higher traffic rates, even such filtering becomes technically problematic. For example, the `mwn-full-packets`, a heavily-loaded Gbps university environment, averages more than 2 TB external traffic each day, with busy-hour loads of 350 Mbps. At that level, it is very difficult to reliably capture the full traffic stream using a simple commodity deployment.

A final issue concerns *using* the captured data. In cases of possible security compromise, it can be of great importance to track down the attacker and assess the damage as quickly as possible. Yet, manually sifting through an immense archive of packet traces to extract a "needle in a haystack" is time-consuming and cumbersome.

In this work we develop a system that uses dynamic packet filtering and buffering

to enable effective bulk-recording of large traffic streams. As this system allows us to conveniently "travel back in time", we term it a *Time Machine*. Our Time Machine buffers network streams first in memory and then on disk, providing several days of nearly-complete (from a forensics and trouble-shooting perspective) historic data and supporting timely access to locate the haystack needles. Our initial application of the Time Machine is as a forensic tool, to extract detailed past information about unusual activities once they are detected. Already the Time Machine has proved operationally useful, enabling diagnosis of a break-in that had gone overlooked at LBNL, whose standard bulk-recorder's static filter had missed capturing the relevant data.

Naturally, the Time Machine cannot buffer an entire high-volume stream. Rather, we exploit the "heavy-tailed" nature of network traffic to partition the stream more effectively (than a static filter can) into a small subset of high interest versus a large remainder of low interest. We then record the small subset and discard the rest. The key insight that makes this work is that 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]. However, very often for forensics and trouble-shooting applications the *beginning* of a large connection contains the most significant information. Put another way, given a choice between recording some connections in their entirety, at the cost of missing others in their entirety; versus recording the beginnings of all connections and the entire contents of most connections, we generally will prefer the latter.

The Time Machine does so using a *cutoff* limit, N: for every connection, it buffers up to the first N bytes of traffic. This greatly reduces the traffic we must buffer while retaining full context for small connections and the beginning for large connections. This simple mechanism is highly efficient: for example, at LBNL, with a cutoff of $N = 20$ KB and a disk storage budget of 90 GB, we can retain 3–5 days of *all* of the site's TCP connections, and, using another 30 GB, 4–6 days for all of its UDP flows (which tend to be less heavy-tailed).

We are not aware of any comparable system for traffic capture. While commercial bulk recorders are available (e.g, *McAfee Security Forensics* [McA]), they appear to use brute-force bulk-recording, requiring huge amounts of disk space. Moreover, due to their black-box nature, evaluating their performance in a systematic fashion is difficult. Another approach, used by many network intrusion detection/prevention systems, is to record those packets that trigger alerts. Some of these systems buffer the start of every connection for a short time (seconds) and store them permanently if the session triggers an alert. Our extension to Bro discussed in Chapter 5 also buffers the first X bytes of each connection until a suitable analyzer is found. However, these systems do not provide long-term buffers or arbitrary access, so they do not support retrospective analysis of a problematic host's earlier activity. Another feature of Bro allows to either record *all* analyzed packets, or *future* traffic once an incident has been detected. Finally, the Packet Vault system was designed to bulk record entire traffic streams [AUH99]. It targets lower data rates and does not employ any filtering.

We organize the remainder of this chapter as follows. In Section 6.2, we briefly summarize the Time Machine's design goals. In Section 6.3, we use trace-driven simulation to explore the feasibility of our approach for data-reduction in three high-volume envi-

ronments. We discuss the Time Machine's architecture in Section 6.4 and present an evaluation of its performance in two of the environments in Section 6.5. Section 6.6 summarizes our work on the system.

6.2 Design Goals

We identified six major design goals for a Time Machine:

- **Provide raw packet data:**
  The Time Machine should enable recording and retrieval of full packets, including payload, rather than condensed versions (e.g., summaries, or just byte streams without headers), in order to prevent losing crucial information.

- **Buffer traffic comprehensively:**
  The Time Machine should manage its stored traffic for time-frames of *multiple days*, rather than seconds or minutes. It should not restrict capture to individual hosts or subnetworks, but keep as widespread data as possible.

- **Prioritize traffic:**
  Inevitably, in high-volume environments we must discard some traffic quickly. Thus, the Time Machine needs to provide means by which the user can express different classes of traffic and the resources associated with each class.

- **Automated resource management:**
  From experience, we know that having to manually manage the disk space associated with high-volume packet tracing becomes tedious and error-prone over time. The Time Machine needs to enable the user to express the resources available to it in high-level terms and then manage these resources automatically.

- **Efficient and flexible retrieval:**
  The Time Machine must support timely queries for different subsets of the buffered data in a flexible and efficient manner. However, its packet capture operation needs to have priority over query processing.

- **Suitable for high-volume environments using commodity hardware:**
  Even though we target large networks with heavily loaded Gbps networks, there is great benefit in a design that enables the Time Machine to run on off-the-shelf hardware, e.g., PCs with 2 GB RAM and 500 GB disk space.

6.3 Feasibility Study

In this section we explore the feasibility of achieving the design goals outlined above by leveraging the heavy-tailed nature of traffic to exclude most of the data in the high-volume streams.

6.3.1 Methodology

To evaluate the memory requirements of a Time Machine, we approximate it using a packet-buffer model. We base our evaluation on connection-level logs from the three environments described below. These logs capture the nature of their environment but with a relatively low volume compared to full packet-level data. Previous work [WDF+05] has shown that we can use flow data to approximate the data rate contributed by a flow, so we can assume that a connection spreads its total traffic across its duration evenly, which seems reasonable for most connections, especially large ones.

We evaluate the packet-buffer model in discrete time steps, enabling us to capture at any point the *volume* of packet data currently stored in the buffer and the *growth-rate* at which that volume is currently increasing. In our simplest simulation, the arrival of a new connection increases the growth-rate by the connection's overall rate (bytes transferred divided by duration); it is decreased by the same amount when it finishes. We then add the notion of keeping data for an extended period of time by introducing an *eviction time* parameter, T_e, which defines how long the buffer stores each connection's data. In accordance with our goals, we aim for a value of T_e on the order of days rather than minutes.

As described so far, the model captures bulk-recording with a timeout but without a *cutoff*. We incorporate the idea of recording only the first N bytes for each connection by adjusting the time at which we decrement the growth-rate due to each connection, no longer using the time at which the connection finishes, but rather the time when it exceeds N bytes (the connection *size cutoff*).

6.3.2 Dataset

We drive our analysis using traces gathered from packet monitors deployed at the Internet access links of the three institutions MWN, LBNL and NERSC.

For our analysis we use connection-level logs of one week from MWN, LBNL, and NERSC. The MWN connection log contains 355 million connections from Monday, Oct. 18, 2004, through the following Sunday. The logs from LBNL and NERSC consist of 22 million and 4 million connections observed in the week after Monday Feb. 7, 2005 and Friday Apr. 29, 2005 respectively.

6.3.3 Analysis of connection size cutoff

As a first step we investigate the heavy-tailed nature of traffic from our environments. Figure 6.1 plots the (empirical) complementary cumulative distribution function (CCDF) of the number of bytes per connection for each of the three environments. Note that a "linear" relationship in such a log-log scaled plot indicates consistency of the tail with a Pareto distribution.

An important consideration when examining these plots is that the data we used – connection summaries produced by the Bro NIDS – are based on the difference in sequence numbers between a TCP connection's SYN and FIN packets. This introduces two forms of *bias*. First, for long-running connections, the NIDS may miss either the

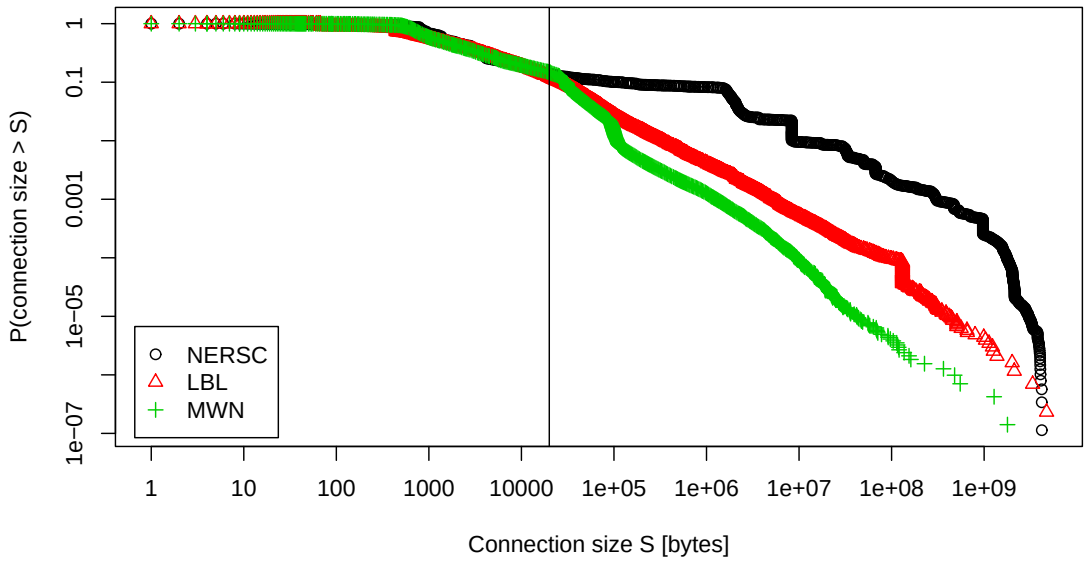


Figure 6.1: Log-log CCDF of connection sizes

initial SYN or the final FIN, thus not reporting a size for the connection. Second, if the connection's size exceeds 4 GB, then the sequence number space will *wrap*; Bro will report only the bottom 32 bits of the size. Both of these biases will tend to *underestimate* the heavy-tailed nature of the traffic, and we know they are significant because the total traffic volume accounted for by the Bro reports is much lower than that surmised via random sampling of the traffic.

The plot already reveals insight about how efficiently a cutoff can serve in terms of reducing the volume of data the Time Machine must store. For a cutoff of 20 KB, corresponding to the vertical line in Figure 6.1, 12% (LBNL), 14% (NERSC) and 15% (MWN) of the connections have a larger total size. The percentage of bytes is much larger, though: 87% for MWN, 96% for LBNL, and 99.86% for NERSC. Accordingly, we can expect a huge benefit from using a cutoff.

Next, using the methodology described above we simulated the packet buffer models based on the full connection logs. Figures 6.2, 6.3 and 6.4 show the required memory for MWN, LBNL, and NERSC, respectively, for different combinations of eviction time T_e and cutoff. A deactivated cutoff corresponds to bulk-recording with a timeout. While the bulk-recording clearly shows the artifacts of time of day and day of week variations, using a cutoff reduces this effect, because we can accompany the cutoff with a much larger timeout, which spreads out the variations. We see that a cutoff of 20 KB quite effectively reduces the buffered volume: at LBNL, with $T_e = 4$ d, the maximum volume, 68 GB, is just a tad higher than the maximum volume, 64 GB, for bulk-recording with $T_e = 3$ h. However, we have increased the duration of data availability by a factor of 32! Note that the volume for simulations with $T_e = 4$ d stops to increase steadily after four days, since starting then connections are being evicted in the buffer model. At

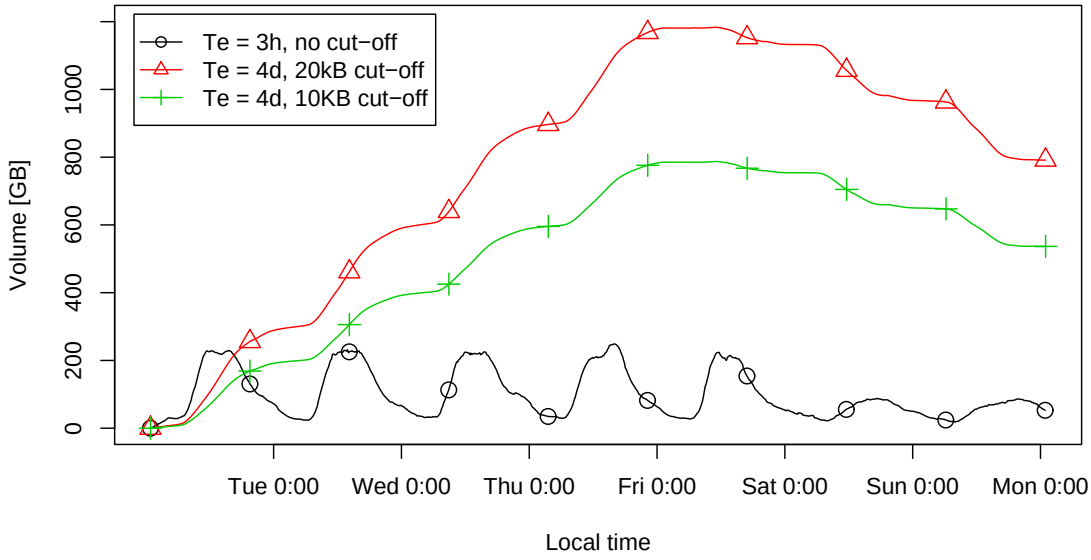


Figure 6.2: Simulated Volume for MWN environment

NERSC, the mean (peak) even decreases from 135 GB (344 GB) to 7.7 GB (14.9 GB). This enormous gain is due to the site's large proportion of high-volume data transfers. As already indicated by the lower fraction of bytes in the larger connections for MWN, the gain from the cutoff is not quite as large, likely due to the larger fraction of HTTP traffic.

Reducing the cutoff by a factor of two further reduces the maximum memory requirements, but only by a factor 1.44 for LBNL, 1.40 for NERSC, and 1.50 for MWN– not by a full factor of two. This is because at this point we are no longer able to further leverage a heavy tail.

The Figures also show that without a cutoff, the volume is spiky. In fact, at NERSC the volume required with $T_e = 1$ h is no more than two times that with $T_e = 1$ m, due to its intermittent bursts. On the other hand, with a cutoff we do not see any significant spikes in the volumes. This suggests that sudden changes in the buffer's growth-rate are caused by a few high-volume connections rather than shifts in the overall number of connections. All in all, the plots indicate that by using a cutoff of 10–20 KB, buffering *several days* of traffic is practical.

6.4 Architecture

The main functions our Time Machine needs to support are *(i)* buffering traffic using a cutoff, *(ii)* migrating (a subset of) the buffered packets to disk and managing the associated storage, *(iii)* providing flexible retrieval of subsets of the packets, and *(iv)* enabling customization. To do so, we use the multi-threaded architecture shown in Figure 6.5, which separates *user interaction* from *recording* to ensure that packet capture

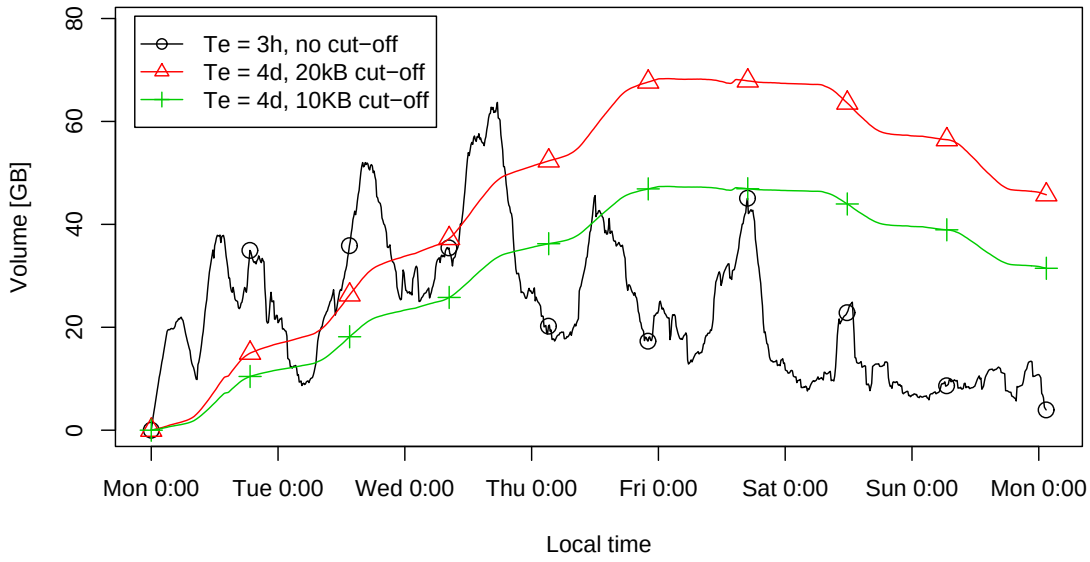


Figure 6.3: Simulated volume for LBNL environment

has higher priority than packet retrieval.

The user interface allows the user to configure the recording parameters and issue queries to the *query processing* unit to retrieve subsets of the recorded packets. The recording thread is responsible for packet capture and storage. The architecture supports customization by splitting the overall storage into several *storage containers*, each of which is responsible for storing a subset of packets within the resources (memory and disk) allocated via the user interface. The *classification* unit decides which packets to assign to each storage container. In addition, the classification unit is responsible for monitoring the cutoff with the help of the *connection tracking* component, which keeps per connection statistics. To enable efficient retrieval, we use an index across all packets stored in all storage containers, managed by the *indexing* module. Finally, access to the packets coming in from the network *tap* is managed by the *capture* unit.

The capture unit receives packets from the network tap and passes them on to the classification unit. Using the connection tracking mechanism, it checks if the connection the packet belongs to has exceeded its cutoff value. If not, it finds the associated storage container, which then stores the packet in memory, indexing it in the process for quick access later on. It later migrates it to disk, and eventually deletes it. Accordingly, the actual Time Machine differs from the connection-level simulation model in that now the buffers are caches that evict *packets* when they are full, rather than evicting whole connections precisely at their eviction time.

Our implementation of the architecture uses the `libpcap` packet capture library [TCP], for which the user can specify a kernel-level BPF [MJ93] capture filter to discard "uninteresting" traffic as early as possible. We collect and store each packet's full content and capture timestamp.

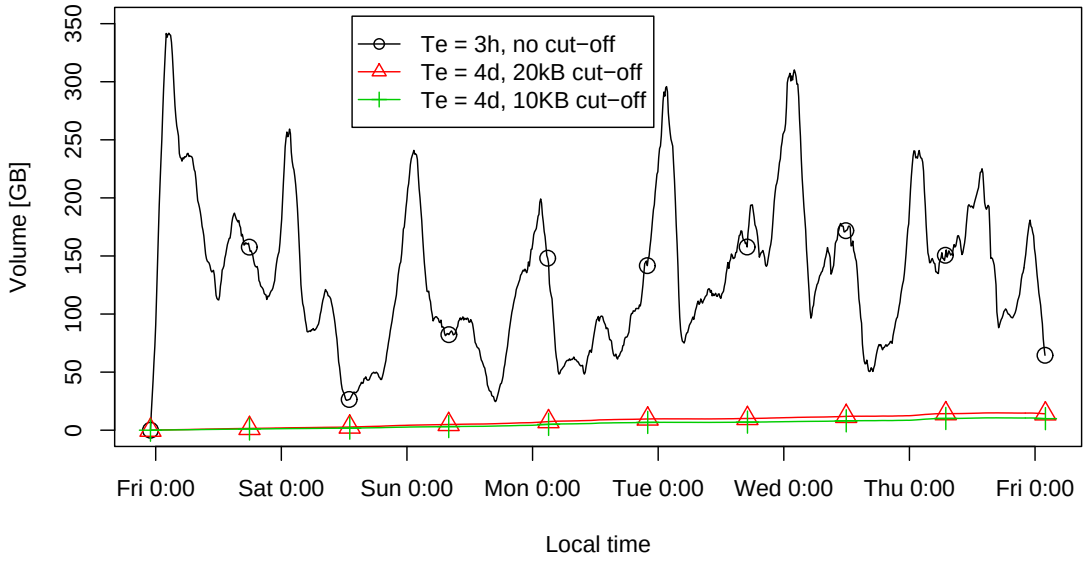


Figure 6.4: Simulated volume for NERSC environment

The capture unit passes the packet to the classification routines, which divide the incoming packet stream into classes according to a user-specified configuration. Each class definition includes a class name, a BPF filter to identify which packets belong to the class, a matching priority, and several storage parameters; for example:

```
class "telnet" { filter "tcp port 23";
  precedence 50; cutoff 10m;
  mem 10m; disk 10g; }
```

which defines a class "`telnet`" that matches, with priority 50, any traffic captured by the BPF filter `"tcp port 23"`. A cutoff of 10 MB is applied, and an in-memory buffer of 10 MB and a disk budget of 10 GB allocated.

For every incoming packet, we look up the class associated with its connection in the connection tracking unit, or, if it is a new connection, match the packet against all of the filters. If more than one filter matches, we assign it to the class with the highest priority. If no filter matches, the packet is discarded.

To track connection cutoffs, the Time Machine keeps state for all active connections in a hash table. If a newly arrived packet belongs to a connection that has exceeded the cutoff limit configured for its class, it is discarded. We manage entries in the connection hash table using a user-configurable inactivity timeout; the timeout is shorter for connections that have not seen more than one packet, which keeps the table from growing too large during scans or denial of service attacks.

For every class, the Time Machine keeps an associated storage container to buffer the packets belonging to the class. Storage containers consist of two ring buffers. The first stores packets in a RAM buffer, while the second buffers packets on disk. The user can

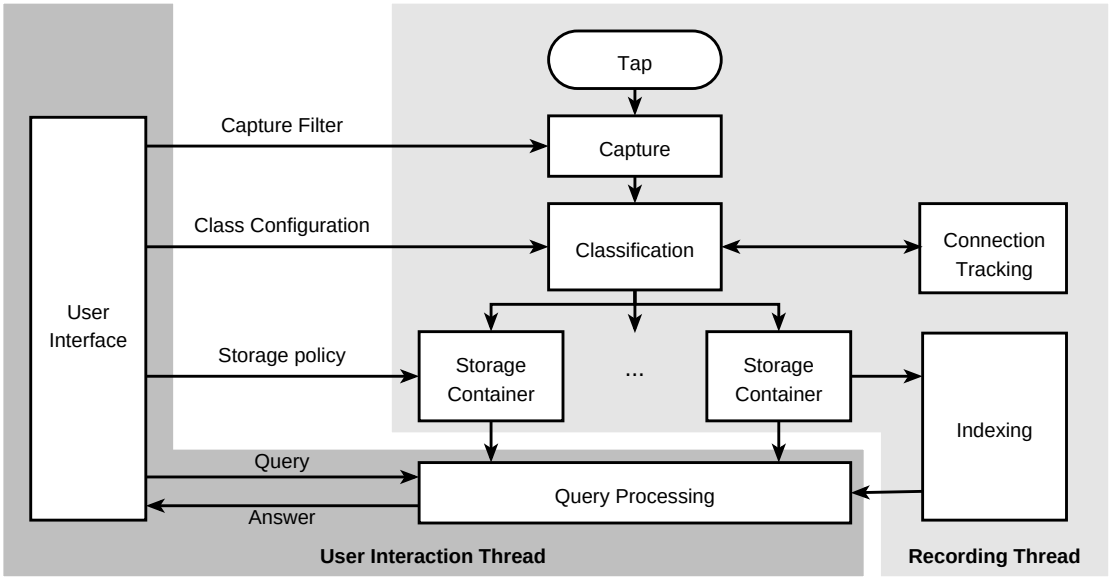


Figure 6.5: Time Machine System Architecture

configure the size of both buffers on a per-class basis. A key motivation for maintaining a RAM buffer in addition to disk storage is to enable near-real-time access to the more recent part of the Time Machine's archive. Packets evicted from the RAM buffer are moved to the disk buffer. We structure the disk buffer as a set of files. Each such file can grow up to a configurable size (typically 10–100s of MB). Once a file reaches this size, we close it and create a new file. We store packets both in memory and on disk in `libpcap` format. This enables easy extraction of `libpcap` traces for later analysis.

To enable quick access to the packets, we maintain *multiple* indexes. The Time Machine is structured internally to support any number of indexes over an arbitrary set of (predefined) protocol header fields. For example, the Time Machine can be compiled to simultaneously support per-address, per-port, and per-connection-tuple indexes. Each index manages a list of time intervals for every unique key value, as observed in the protocol header field (or fields) of the packets. These time intervals provide information on whether packets with that key value are available in a given storage container and at what starting timestamp, enabling fast retrieval of packets. Every time the Time Machine stores a new packet it updates each associated index. If the packet's key – a header field or combination of fields – is not yet in the index, we create a new entry containing a zero-length time interval starting with the timestamp of the packet. If an entry exists, we update it by either extending the time interval up to the timestamp of the current packet, or by starting a new time interval, if the time difference between the last entry in the existing interval and the new timestamp exceeds a user-defined parameter. Thus, this parameter trades off the size of the index (in terms of number of intervals we maintain) for how precisely a given index entry localizes the packets of interest within a given storage container. As interval entries age, we migrate them from in-memory index

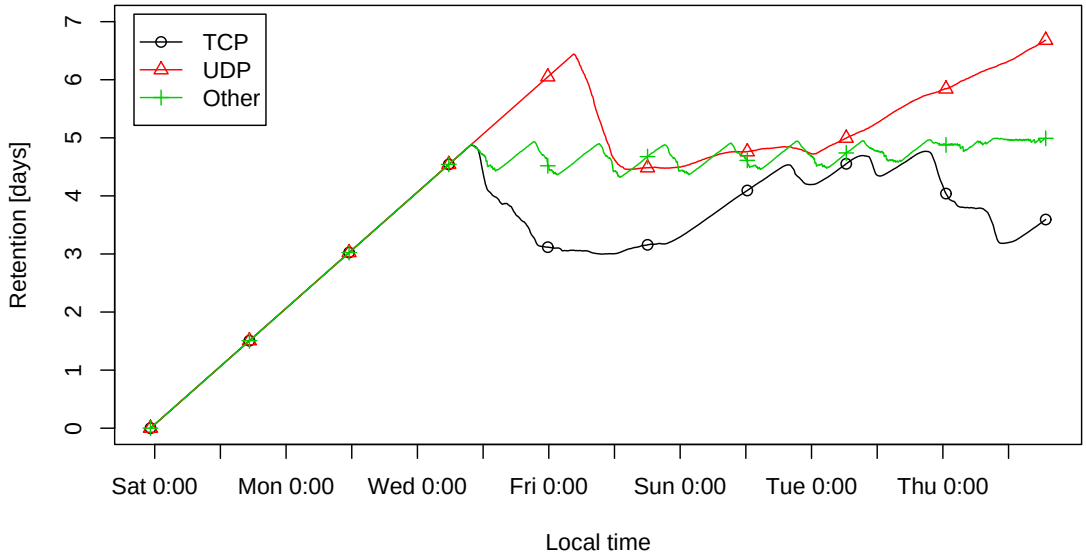


Figure 6.6: Retention in the LBNL environment

structures to index files on disk, doing so at the same time the corresponding packets in the storage containers migrate from RAM to disk. In addition, the user can set an upper limit for the size of the in-memory index data structure.

The final part of the architecture concerns how to find packets of interest in the potentially immense archive. While this can be done using brute force (e.g., running `tcpdump` over all of the on-disk files), doing so can take a great deal of time, and also have a deleterious effect on Time Machine performance due to contention for the disk. We address this issue using the query-processing unit, which provides a flexible language to express queries for subsets of the packets. Each query consists of a logical combination of time ranges, keys, and an optional BPF filter. The query processor first looks up the appropriate time intervals for the specified key values in the indexing structures, trimming these to the time range of the query. The logical *or* of two keys is realized as the union of the set of intervals for the two keys, and an *and* by the intersection. The resulting time intervals correspond to the time ranges in which the queried packets originally arrived. We then locate the time intervals in the storage containers using binary search. Since the indexes are based on time intervals, these only limit the amount of data that has to be scanned, rather then providing exact matches; yet this narrowing suffices to greatly reduce the search space, and by foregoing exact matches we can keep the indexes much smaller. Accordingly, the last step consists of scanning all packets in the identified time ranges and checking if they match the key, as well as an additional BPF filter if supplied with the query, writing the results to a `tcpdump` trace file on disk.

6.5 Evaluation

To evaluate the Time Machine design, we ran an implementation at two of the sites discussed in Section 6.3.2. For LBNL, we used three classes, each with a 20 KB cutoff: TCP traffic, with a space budget of 90 GB; UDP, with 30 GB; and Other, with 10 GB. To evaluate the "hindsight" capabilities, we determine the *retention*, i.e., the distance back in time to which we can travel at any particular moment, as illustrated in Figure 6.6. Note how retention increases after the Time Machine starts until the disk buffers have filled. After this point, retention correlates with the incoming bandwidth for each class and its variations due to diurnal and weekly effects. New data forces the eviction of old data, as shown for example by the retention of TCP shortening as the lower level weekend traffic becomes evicted around Wed–Thu. The TCP buffer of 90 GB allows us to retain data for 3–5 days, roughly matching the predictions from the LBNL simulations (recall the volume biases of the connection-level data discussed in Section 6.3). Use of a cutoff is highly efficient: on average, 98% of the traffic gets discarded, with the remainder imposing an average rate of 300 KB/s and a maximum rate of 2.6 MB/s on the storage system. Over the 2 weeks of operation, `libpcap` reported only 0.016% of all packets dropped.

Note that classes do not have to be configured to yield an identical retention time. The user may define classes based on their view of utility of having the matching traffic available in terms of cutoff and how long to keep it. For example we might have included a class configuration similar to the example in Section 6.4 in order to keep more of Telnet connections for a longer period of time.

Operationally, the Time Machine has already enabled the diagnosis of a break-in at LBNL by having retained the response to an HTTP request that was only investigated three days later. The Time Machine's data both confirmed a successful compromise and provided additional forensic information in terms of the attacker's other activities. Without the Time Machine, this would not have been possible, as the site cannot afford to record its full HTTP traffic for any significant length of time.

At MWN we ran preliminary tests of the Time Machine, but we have not yet evaluated the retention capability systematically. First results show that about 85% of the traffic gets discarded, with resulting storage rates of 3.5 (13.9) MB/s average (maximum). It appears that the larger volume of HTTP traffic is the culprit for this difference compared to LBNL, due to its lesser heavy-tailed nature; this matches the results of the MWN connection-level simulation. For this environment it seems we will need to more aggressively exploit the classification and cutoff mechanisms to appropriately manage the large fraction of HTTP traffic.

The fractions of discarded traffic for both LBNL and MWN match our predictions well, and the resulting storage rates are reasonable for today's disk systems, as demonstrated in practice. The connection tracking and indexing mechanisms coped well with the characteristics of real Internet traffic. We have not yet evaluated the Time Machine at NERSC, but the simulations promise good results.

6.6 Summary

In this chapter, we introduce the concept of a *Time Machine* for efficient network packet recording and retrieval. The Time Machine can buffer several days of raw high-volume traffic using commodity hardware. It provides an efficient query interface to retrieve the packets in a timely fashion, and automatically manages its available storage. The Time Machine relies on the simple but crucial observation that due to the "heavy-tailed" nature of network traffic, we can record most connections in their entirety, yet skip the bulk of the total volume, by storing up to (a customizable) cutoff limit of bytes per connection. We have demonstrated the effectiveness of the approach using a trace-driven simulation as well as operational experience with the actual implementation in two environments. A cutoff of 20 KB increases data availability from several hours to several days when compared to brute-force bulk recording.

In operational use, the Time Machine has already proved valuable by enabling diagnosis of a break-in that standard bulk-recording had missed. That means, that overall, the Time Machine provides a valuable tool for comlementing NIDS with its value for detailed network forensics. In future work, we intend to add a remote access interface to enable real-time queries for historic network data by NIDS.

7 Conclusion

7.1 Summary

At a first glance the feature-set and the analysis capabilities of modern Network Intrusion Detection Systems are stunning. As soon as such a system is deployed operationally in a high-volume network environment, it becomes quickly clear, that some analysis capabilities are not always usable in practice: The system cannot perform the desired analysis fast enough to keep up with the network traffic.

Thus, high-volume network environments are quite a challenge for today's NIDS. On the example of the Bro NIDS, we demonstrate, that a configurable resource management is as vital for a NIDS as a large spectrum of detection capabilities. We explore tradeoffs in the operation of NIDS between resource usage and analysis that span a range from predictive to adaptive to retrospective. We explore the predictive aspect by developing a model of the resource consumption of the Bro NIDS depending on network traffic characteristics. Based on this model, we are able to predict the resource consumption if we know the trend of the network traffic. We address the adaptive aspect in designing, implementing and evaluating a framework for NIDS that allows a flexible, dynamic analysis. What analysis is performed can be decided per individual connection. Therefore, the framework is suitable for implementing mechanisms to dynamically decide what analysis to perform and to what detail. The retrospective aspect is the focus of a system to record full packet traces suitable for offline deep security analysis.

We discuss in Chapter 3, that the operational deployment of NIDS in high-volume network environments raises severe resource management issues. While it is highly desirable to have systems that are robust against evasion attacks, in high-volume network environments it is not feasible to counter these attacks by accumulating and keeping state for an indefinitely long time. We extend Bro's state management with new timeouts and state expiration optimizations. Furthermore, we identify a set of pitfalls in implementing intrusion detection systems that, in high-volume environments, will lead to overloading or crashing the system sooner or later. If configured properly, Bro stuffed with our extensions described in Chapter 3 is fit to run continuously in our three high-volume network environments.

For modeling the resource usage of the Bro NIDS depending on the network traffic characteristics in Chapter 4, we first performed a detailed code analysis. We split the system into smaller components, each of which has different tasks and contributions to the overall analysis. For these components, we measure the resource consumption on real-world traces separately and extrapolate from the individual measurements the resource usage of the whole system. We develop a method using random sampling on a per-connection basis to perform the measurements of the components in acceptable time

and with acceptable disk space resources. We evaluate the fundamental problems that come along with the random connection sampling approach and propose techniques to compensate for the measurement error. As result, we have a tool that supports a NIDS operator in evaluating which of a set of configuration options makes the optimal use out of the resources at hand. This tool evaluates NIDS resource usage only for a rather short time interval (minutes or hours). Therefore, we developed mechanisms to predict the resource usage of the NIDS Bro and a given configuration on the basis of connection-level aggregated data. This technique can be used both to evaluate and compare the resource usage of different configurations over a longer time (e.g., a week) and to predict the resource requirements of a given Bro configuration if the network traffic characteristics change.

In Chapters 3 and 4, we have seen that besides variable analysis capabilities it is also vital for a NIDS to be able to use these capabilities dynamically. In Chapter 5 we propose a framework for NIDS that allows to dynamically decide per connection what analysis to perform. Our primary use for this is to detect the application layer protocol for each connection and perform the corresponding analysis. This approach is necessary as more and more applications resist a classification via the used transport layer protocol port. Currently we use byte-level signatures to detect specific application layer protocols, but the framework is flexible enough to allow arbitrary protocol detection mechanisms to be implemented. Being able to perform detailed protocol analysis (using Bro's existing analyzers) on connections running on non standard ports we are able to detect, e.g., IRC botnets or FTP servers offering pirated content. Regarding resource usage, the new framework does not impose an overhead to the system. However protocol detection is expensive. After all, each packet needs to be checked with a protocol detection mechanism.

Often a NIDS alarm does not offer much context to the operator. However, often it would be extremely helpful to know for example what happened to the victim before the alert or whether the attacker did anything suspicious before the alarm was raised. The system we describe in Chapter 6 allows to bulk record a full packet trace of most traffic that is interesting for such post-alert examinations. We leverage that it is usually not worthwhile to record long connections (e.g., ftp bulk data transfers) in their entirety. Given the heavy tailed distribution of connection sizes, our system is able to record a large fraction (up to 99%) of all connections completely. For the remaining fraction, it retains at least the connection beginnings up to a cutoff value of N bytes. In one of our loaded gigabit network environments, the system was able to accommodate up to 5 days of such "compressed" packet trace in a disk budget of only 130 GB.

7.2 Outlook

For future work, different aspects worked out in this thesis can be extended. Our model of the Bro resource usage described in Chapter 4 can be fleshed out further, to comprise the memory usage for user-level state. Supported with a more fine-grained instrumentation for measuring memory consumption of the single components, this will allow to

extrapolate the memory usage for complex Bro configurations more exactly. The extended model will also make predictions on memory usage based on connection level data possible.

Regarding our dynamic analysis framework presented in Chapter 5 we see great potential in this approach beyond our discussed application. Since it is possible to use different analyzers in parallel and in sequence for arbitrary connections, we believe, that the framework can be used for a host of NIDS analysis applications. An example is another approach to resource management: Based on the available resources at the moment and some sort of "screening test" for each connection, one could enable a NIDS to dynamically decide whether the connection is worthwhile to be analyzed.

Our short-time motivation for developing the Time Machine was network security forensics. However, we designed the system as a supplementary tool for Network Intrusion Detection. Currently, a communication interface between the Time Machine and Bro is under development. As possible application we picture the NIDS to request data of connections, that were recognized to be suspicious but not fully analyzed due to CPU resource limits. Another application is to let the NIDS automatically request and closely analyze all packets that were exchanged with a host that is detected to be a scanner.

As medium-range goal, we plan to combine our approaches. By integrating our resource usage prediction mechanisms into a NIDS, the NIDS is enabled to self adjust the resource/analysis tradeoffs dynamically. A NIDS that is able to determine the resource usage by different analyses of the current traffic online can dynamically adapt the analysis depth at any point in time. Thus it can always use the available resources to full capacity. With the additional help of the Time Machine, it can even do more than dynamically adjusting analysis depth to the current workload. During peak time the NIDS can limit its analysis depth or breadth but instruct the Time Machine to record the traffic which itself ignores with a larger cutoff value. The NIDS can then use times (e.g., night times) with lower traffic volume to rework traffic that was not analyzed, but recorded, during past peak times.

Our efforts target the long-range goal for operational network intrusion detection: To supply the operator with exactly the right quantity of high-quality information on policy violations. While the mechanisms and approaches developed in this thesis are important steps towards this long-range goal, there are still many problems of NIDS and NIPS open. A challenge is to understand and examine the tradeoffs between resource usage and analysis of distributed NIDS. On the one hand, a major goal of deploying distributed NIDS is to reduce resource usage on the single sensors. On the other hand the distributed character introduces a new class of resource usage: the workload that is caused by the communication and correlation necessary in these systems. In this context, an interesting field of tradeoffs that deserves closer exploration is how local aggregation, communication and analysis capabilities interact.

A Contribution of Single Analyzers to CPU Load

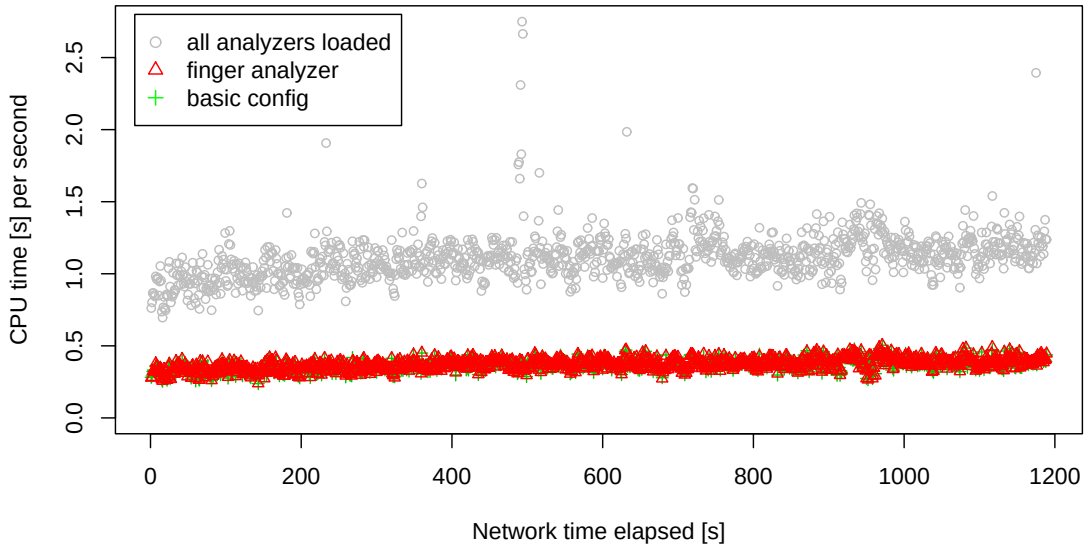


Figure A.1: Contribution of Finger analyzer to CPU load

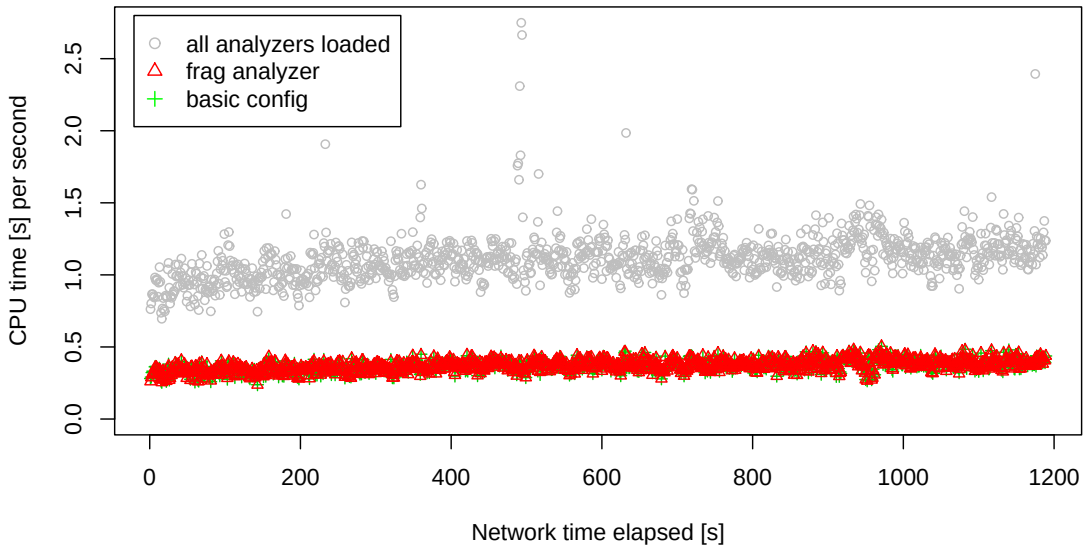


Figure A.2: Contribution of frag analyzer to CPU load

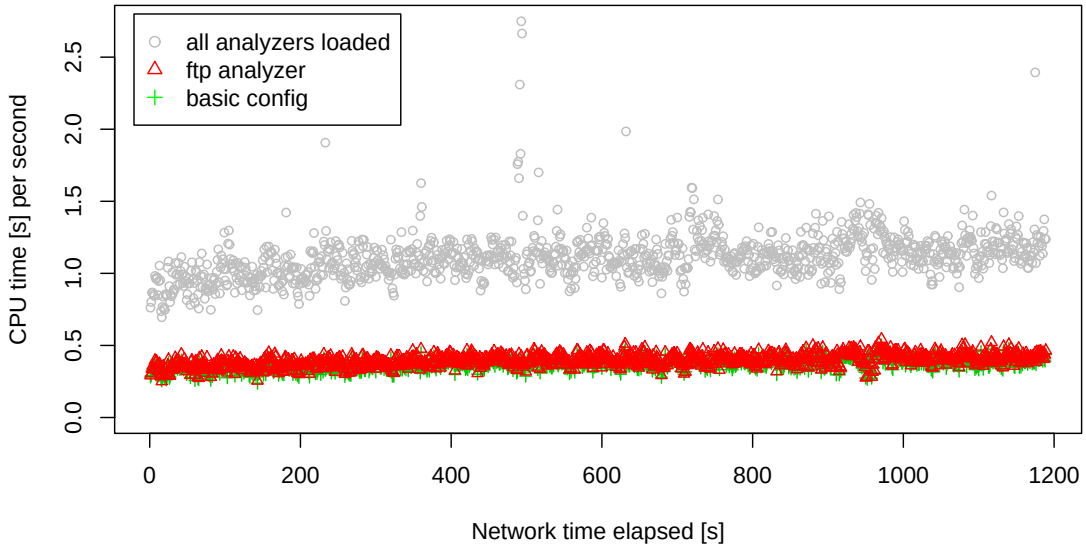


Figure A.3: Contribution of FTP analyzer to CPU load

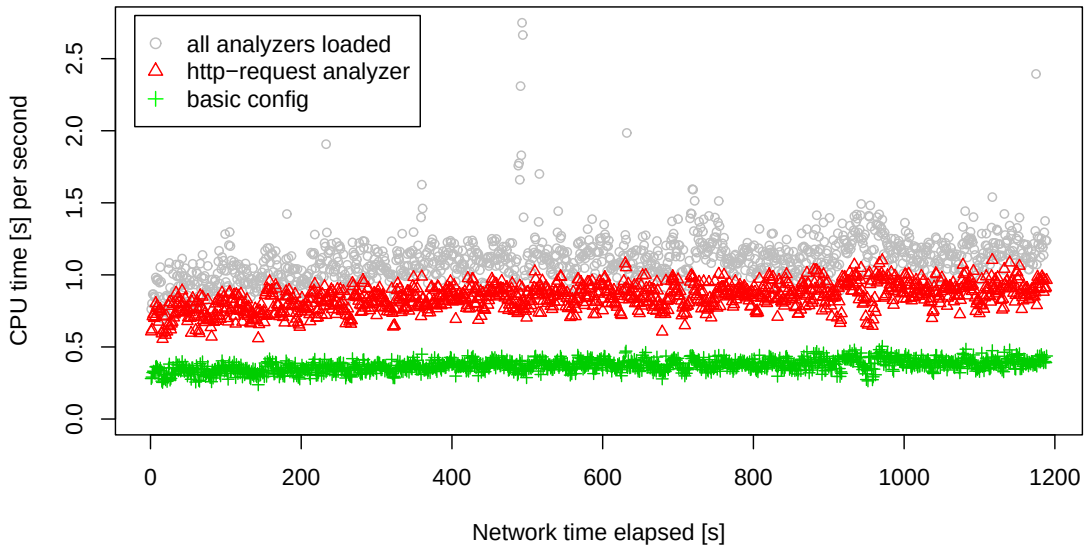


Figure A.4: Contribution of HTTP-request analyzer to CPU load

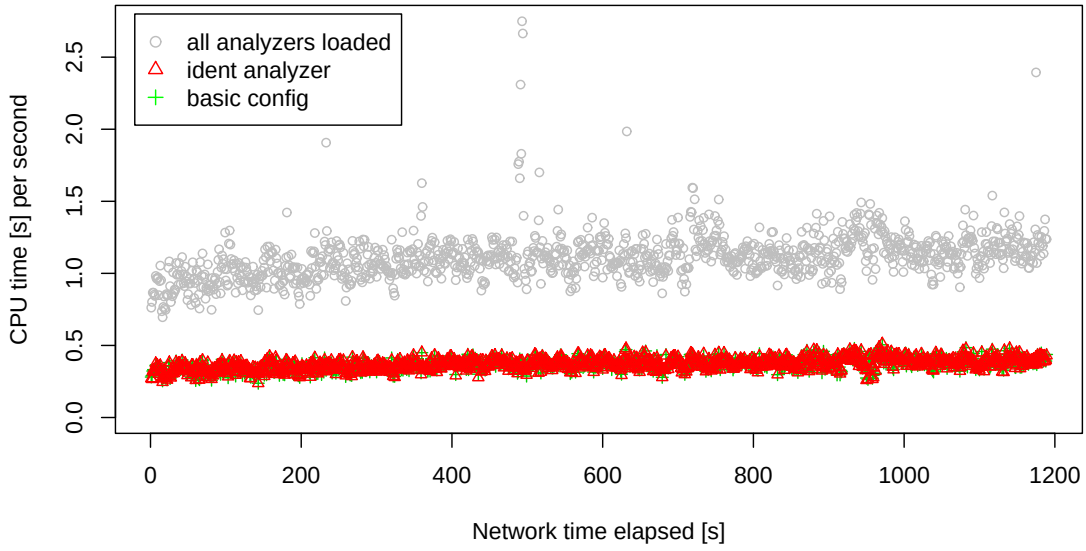


Figure A.5: Contribution of ident analyzer to CPU load

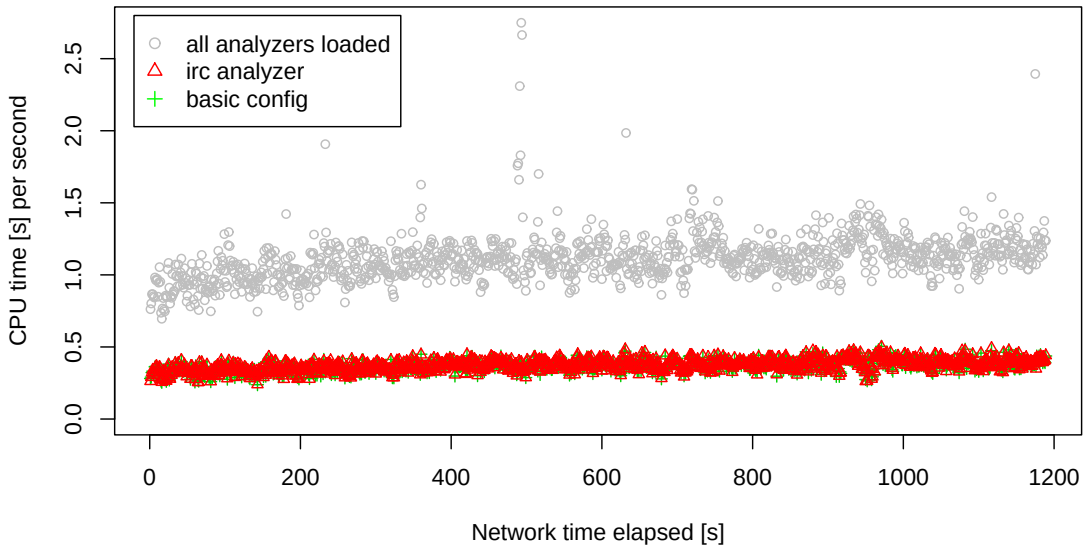


Figure A.6: Contribution of IRC analyzer to CPU load

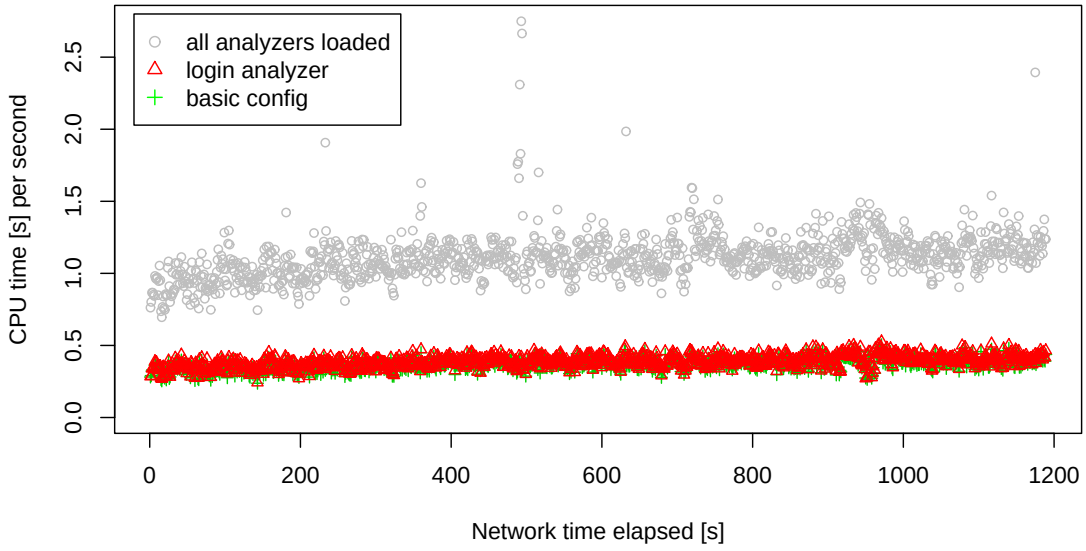


Figure A.7: Contribution of login analyzer to CPU load

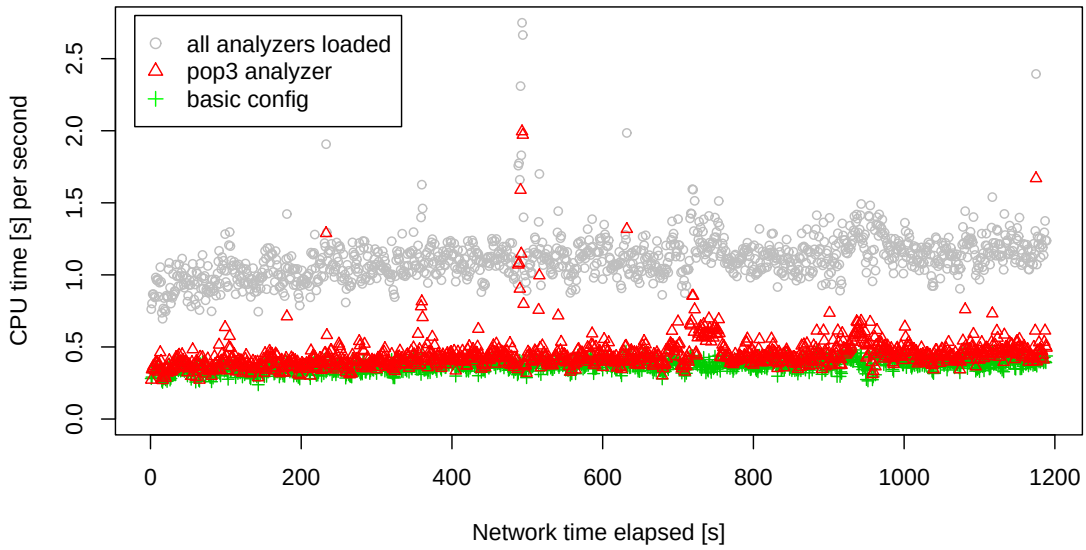


Figure A.8: Contribution of POP3 analyzer to CPU load

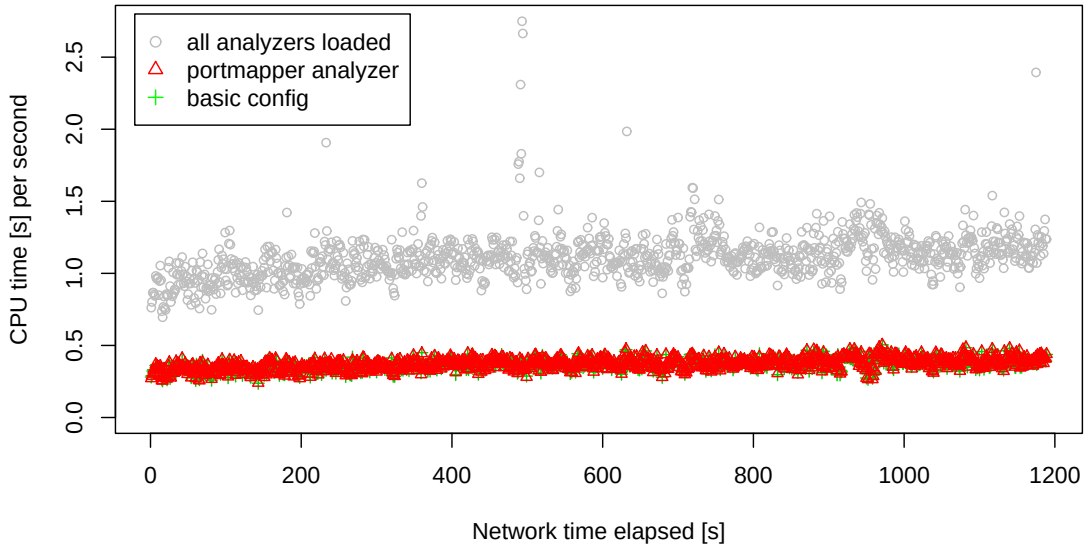


Figure A.9: Contribution of portmapper analyzer to CPU load

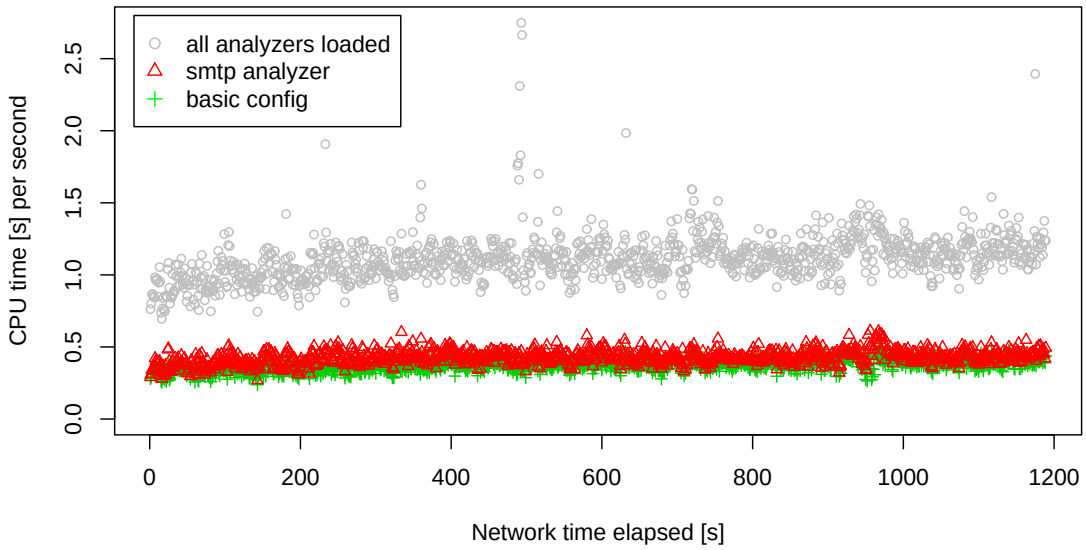


Figure A.10: Contribution of SMTP analyzer to CPU load

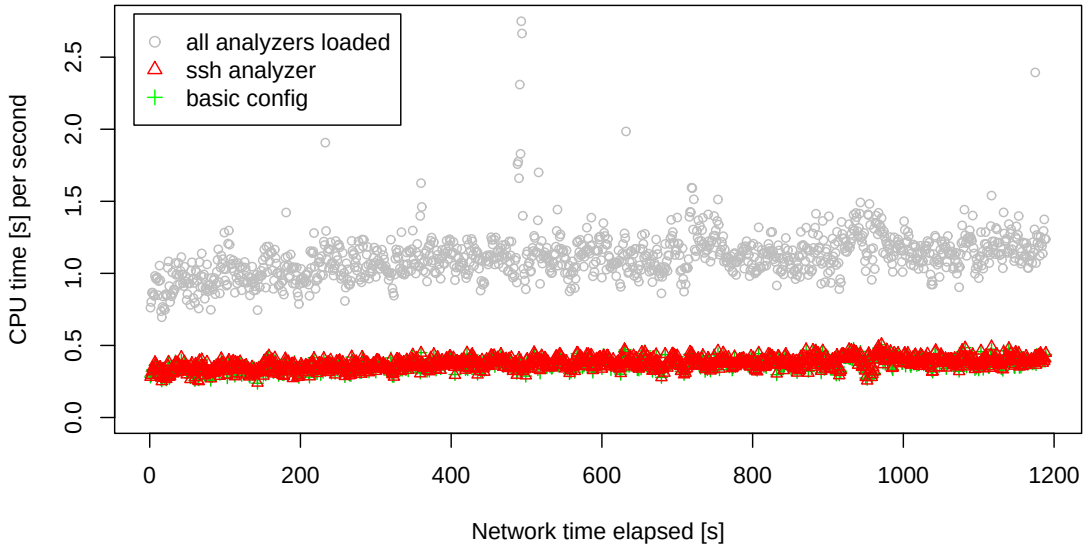


Figure A.11: Contribution of ssh analyzer to CPU load

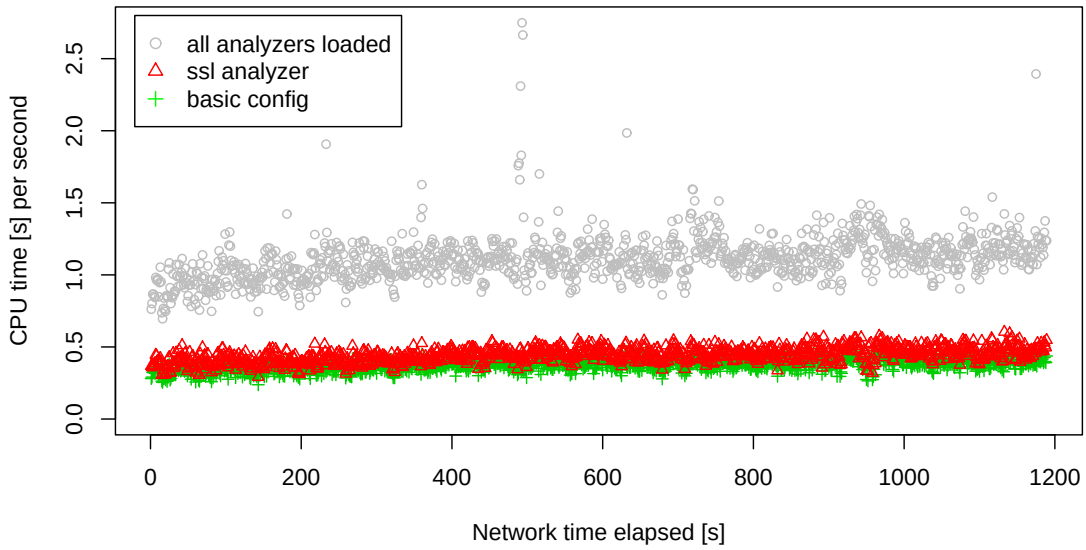


Figure A.12: Contribution of SSL analyzer to CPU load

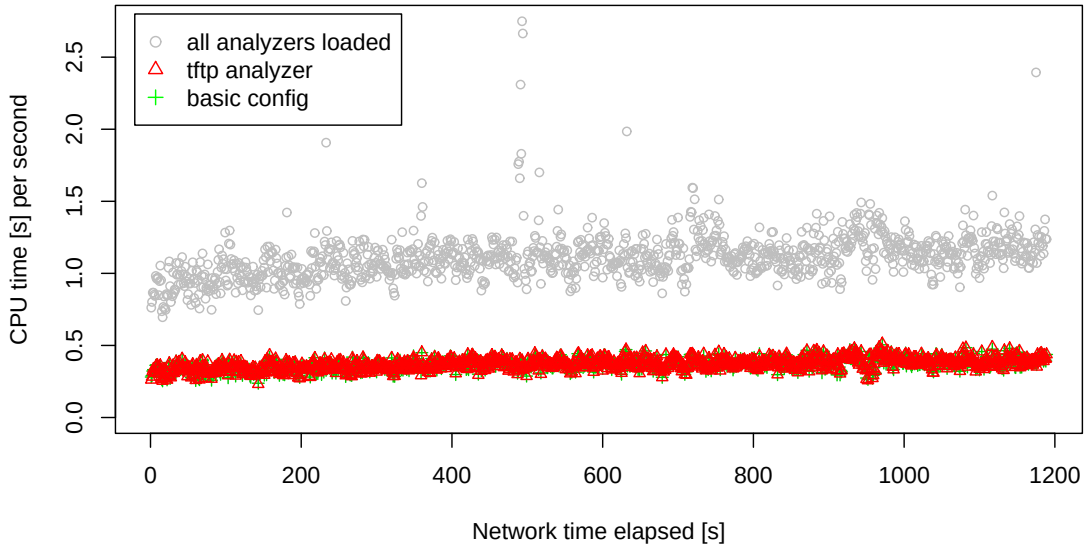


Figure A.13: Contribution of TFTP analyzer to CPU load

Bibliography

[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.

[AUH99] C.J. Antonelli, M. Undy, and P. Honeyman. The Packet Vault: Secure Storage of Network Data. In *Proc. Workshop on Intrusion Detection and Network Monitoring*, pages 103–110, April 1999.

[Axe99] 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.

[Bej04] Richard Bejtlich. *The Tao of Network Security Monitoring*. Addison-Wesley, 2004.

[Bla] CERT Advisory CA-2003-20 W32/Blaster worm. `http://www.cert.org/advisories/CA-2003-20.html`.

[Bld] BleedingSnort. `http://bleedingsnort.com`.

[BS06] Salman A. Baset and Henning Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. In *Proc. IEEE Infocom 2006*, 2006.

[BT] BitTorrent. `http://www.bittorrent.com`.

[CB98] Patrick Crowley and Jean-Loup Baer. On the use of trace sampling for architectural studies of desktop applications. Technical report, University of Washington, 1998.

[CBW96] Roderick Chapman, Alan Burns, and Andy Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.

[CKY+04] T.S. Choi, C.H. Kim, S. Yoon, J.S. Park, B.J. Lee, H.H. Kim, H.S. Chung, and T.S. Jeong. Content-aware Internet Application Traffic Measurement and Analysis. In *Proc. Network Operations and Management Symposium*, 2004.

[Cla] Clam AntiVirus. `http://www.clamav.net`.

[CW03] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proc. USENIX Security Symposium*, 2003.

[DCA+03] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin M. Vahdat. Model-based resource provisioning in a web service utility. In *Proc. USENIX Symposium on Internet Technologies and Systems*, 2003.

[Der03] Luca Deri. Improving Passive Packet Capture: Beyond Device Polling. Technical report, University of Pisa, 2003.

[DI89] M. V. Devarakonda and R. K. Iyer. Predictability of process resource usage: a measurement-based study on unix. *IEEE Transactions on Software Engineering*, 15(12):1579–1586, 1989.

[Din98] P. A. Dinda. The statistical properties of host load. Technical report, Carnegie Mellon University, 1998.

[Din01] P. A. Dinda. Online prediction of the running time of tasks. In *Proc. ACM Conference on Measurement and Modeling of Computer Systems - SIGMETRICS*, 2001.

[DKPS05] Holger Dreger, Christian Kreibich, Vern Paxson, and Robin Sommer. Enhancing the accuracy of network-based intrusion detection with host-based context. In *Proc. Intrusion and Malware Detection and Vulnerability Assessment (DIMVA)*, 2005.

[DO00] P. A. Dinda and D. R. O'Hallaron. Host load prediction using linear models. *Cluster Computing*, 3(4):265–280, 2000.

[DP05] S. Dharmapurikar and V. Paxson. Robust TCP Stream Reassembly In the Presence of Adversaries. In *Proc. USENIX Security Symposium*, 2005.

[DSn] DSniff. `www.monkey.org/~dugsong/dsniff`.

[DWF03] C. Dewes, A. Wichmann, and A. Feldmann. An Analysis Of Internet Chat Systems. In *Proc. ACM Internet Measurement Conference*, 2003.

[EBR03] J.P. Early, C.E. Brodley, and C. Rosenberg. Behavioral Authentication of Server Flows. In *Proc. Annual Computer Security Applications Conference*, 2003.

[End] ENDACE measurement systems. `http://www.endace.com`.

[Ent] Enterasys Networks, Inc. Enterasys Dragon. `http://www.enterasys.com/products/ids`.

[ETF03] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Effects of clock resolution on the scheduling of interactive and soft real-time processes. In *Proc. ACM Conference on Measurement and Modeling of Computer Systems - SIGMETRICS*, 2003.

[Fac96] Patrick Facchini. A statistical method for real-time software estimation. Master's thesis, Linköping University, Sweden, 1996.

[FBR06] D. Fliegl, T. Baur, and H. Reiser. Nat-O-Mat: Ein generisches Intrusion Prevention System. In *Proc. 20. DFN-Arbeitstagung <FC>ber Kommunikationsnetze*, 2006.

[FC] Freechal P2P. `http://www.freechal.com`.

[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 Conference*, 1998.

[FP01] Sally Floyd and Vern Paxson. Difficulties in Simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4), 2001.

[gpr] Gnu binutils. `http://www.gnu.org/software/binutils/`.

[HK98] C-H. Hsu and U. Kremer. A framework for qualitative performance prediction. Technical report, Department of Computer Science, Rutgers University, July 1998.

[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.

[Hon05] The honeynet project & research alliance: Know your enemy: Tracking botnets. `http://www.honeynet.org/papers/bots`, 2005.

[HSSW05] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: Automated Construction of Application Signatures. In *Proc. ACM Workshop on Mining Network Data*, 2005.

[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.

[IAN] The Internet Corporation for Assigned Names and Numbers. `http://www.iana.org`.

[Int] McAfee IntruShield Network IPS Appliances. `http://www.networkassociates.com`.

[IPP] The IPP2P project. `http://www.ipp2p.org/`.

[ISS] Internet security systems (iss). `http://www.iss.net/`.

[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.

[JU01] John E. Gaffney Jr and Jacob W. Ulvila. Evaluation of intrusion detectors: A decision theory approach. In *Proc. IEEE Symposium on Security and Privacy*, page 50, Washington, DC, USA, 2001. IEEE Computer Society.

[Kal00] C. Kalt. Internet Relay Chat: Client Protocol. RFC 2812, 2000.

[KFB99] Nirav H. Kapadia, Jose A. B. Fortes, and Carla E. Brodley. Predictive application-performance modeling in a computational grid environment. In *Proc. IEEE Symposium on High Performance Distributed Computing*, 1999.

[KM95] K.Shanmugam and A. D. Malony. Performance extrapolation of parallel programs. In *Proc. 24th International Conference on Parallel Processing*, 1995.

[KPF05] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *Proc. ACM SIGCOMM*, 2005.

[KPW05] A. Kumar, V. Paxson, and N. Weaver. Exploiting underlying structure for detailed reconstruction of an internet scale event. In *Proc. Internet Measurement Conference*, 2005.

[KS05] Christian Kreibich and Robin Sommer. Policy-controlled event management for distributed intrusion detection. In *Proc. International Workshop on Distributed Event-Based Systems (DEBS)*, 2005.

[KUE+00] Tahsin Kurc, Mustafa Uysal, Hyeonsang Eom, Jeff Hollingsworth, Joel Saltz, and Alan Sussman. Efficient performance prediction for large-scale data-intensive applications. *The International Journal of High Performance Computing Applications*, 14(3):216–227, 2000.

[KV03] Christopher Kruegel and Giovanni Vigna. Anomaly Detection of Web-based Attacks. In *Proc. ACM Conference on Computer and Communications Security*, 2003.

[L7] Application Layer Packet Classifier for Linux. `http://l7-filter.sourceforge.net`.

[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.

[LFM+02] Wenke Lee, Wei Fan, Matthew Miller, Salvatore J. Stolfo, and Erez Zadok. Toward cost-sensitive modeling for intrusion detection and response. *Journal of Computer Security*, 10(1-2):5–22, 2002.

[LHS98] J. Landrum, J. Hardwick, and Q. Stout. Predicting algorithm performance. *Computing Science and Statistics*, 30:309–314, 1998.

[Lib] libpcap. `http://www.tcpdump.org`.

[LS00] Christian A. Lang and Ambuj K. Singh. Performance prediction of high-dimensional index structures using sampling. Technical report, Univ. of California at Santa Barbara, 2000.

[LTWW93] Will Leland, Murad Taqqu, Walter Willinger, and Daniel Wilson. On the self-similar nature of ethernet traffic. In *Proc. ACM SIGCOMM*, 1993.

[Mag] libmagic — Magic Number Recognition Library.

[Mai05] M. Mai. Dynamic Protocol Analysis for Network Intrusion Detection Systems. Master's thesis, TU M<FC>nchen, 2005.

[McA] McAfee. McAfee Security Forensics. `http://www.mcafeesecurity.com/us/products/mcafee/forensics/security_forensics.htm`.

[Men93] C. L. Mendes. Performance prediction by trace transformation. In *Brazilian Symposium on Computer Architecture*, 1993.

[MJ93] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. Winter USENIX Conference*, 1993.

[MMC04] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proc. ACM Conference on Measurement and Modeling of Computer Systems - SIGMETRICS*, 2004.

[MMS95] B.W. Mohr, A.D. Malony, and K. Shanmugam. Speedy: An integrated performance extrapolation tool for pc++ programs. In *Proc. Joint Conf. Performance Tools and MMB*, 1995.

[MP05] A.W. Moore and K. Papagiannaki. Toward the Accurate Identification of Network Applications. In *Proc. Passive and Active Measurement Workshop*, 2005.

[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.

[MT93] S. McCanne and C. Torek. A randomized sampling clock for cpu utilization estimation and code profiling. In *Proc. USENIX Winter Conference*, 1993.

[MVS01] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet Denial-of-Service Activity. In *Proc. USENIX Security Symposium*, 2001.

[MZ05] A.W. Moore and D. Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques. In *Proc. ACM SIGMETRICS*, 2005.

[NF] Linux NetFilter. `http://www.netfilter.org`.

[Pax94] Vern Paxson. Empirically-Derived Analytic Models of Wide-Area TCP Connections. *IEEE/ACM Transactions on Networking*, 2(4), 1994.

[Pax99] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[PF95] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–224, June 1995.

[PPSP06] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for writing application protocol parsers. In *Proc. Internet Measurement Conference*, 2006.

[PTN98] Thomas H. Ptacek and Newsham Timothy N. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.

[Ran01] Marcus J. Ranum. Experiences Benchmarking Intrusion Detection Systems. Technical report, NFR Security, Inc., 2001.

[Roe99] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. Systems Administration Conference*, 1999.

[RSP] Configuring SPAN and RSPAN (Cisco Catalyst 6500 Series). `http://www.cisco.com/univercd/cc/td/doc/product/lan/cat6000/sw_7_5/confg_gd/span.pdf`.

[RSSD04] M. Roughan, S. Sen, O. Spatscheck, and N.G. Duffield. Class of Service Mapping for QoS: A Statistical Signature Based Approach To IP Traffic Classification. In *Proc. ACM Internet Measurement Conference*, 2004.

[SBC06] P. Shivam, S. Babu, and J. Chase. Learning application models for utility resource planning. In *Proc. IEEE Conference on Autonomic Computing*, 2006.

[SBS96] R. H. Saavedra-Barrera and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, 1996.

[Sch05] Fabian Schneider. Performance evaluation of packet capturing systems for high-speed networks. Master's thesis, Technische Universität München, November 2005.

[Sha89] A. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[SM04] Colleen Shannon and David Moore. The spread of the witty worm. `http://www.caida.org/analysis/security/witty`, 2004.

[Sno] Snot. `http://www.stolenshoes.net/sniph/index.html`.

[Som05] R. Sommer. *Viable Network Intrusion Detection in High-Performance Environments*. PhD thesis, TU M<FC>nchen, 2005.

[SP03a] Umesh Shankar and Vern Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *Proc. IEEE Symposium on Security and Privacy*, 2003.

[SP03b] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proc. ACM Conference on Computer and Communications Security*, 2003.

[SP05] Robin Sommer and Vern Paxson. Exploiting independent state for network intrusion detection. In *Proc. of the 21st Annual Computer Security Applications Conference (ACSAC)*, pages 59–71, Washington, DC, USA, 2005. IEEE Computer Society.

[SPW02] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in Your Spare Time. In *Proc. USENIX Security Symposium*, 2002.

[SS05] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *Proc. USENIX Symposium on Networked Systems Design & Implementation*, 2005.

[SSW04] S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. In *Proc. World Wide Web Conference*, 2004.

[Sti] Stick. `http://packetstormsecurity.nl/distributed/stick.htm`.

[TC97] K.M.C. Tan and B.S. Collie. Detection and classification of TCP/IP network services. In *Proc. Annual Computer Security Applications Conference*, 1997.

[TCP] `tcpdump`. `http://www.tcpdump.org`.

[Tol95] Sivan A. Toledo. *Quantitative Performance Modeling of Scientific Computations and Creating Locality in Numerical Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1995.

[UPS+05] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proc. ACM Conference on Measurement and Modeling of Computer Systems - SIGMETRICS*, 2005.

[Val] `valgrind`. `http://developer.kde.org/~sewardj`.

[WDF+05] J. Wallerich, H. Dreger, A. Feldmann, B. Krishnamurthy, and W. Willinger. A Methodology for Studying Persistency Aspects of Internet Flows. *ACM SIGCOMM Computer Communication Review*, 35(2):23–36, April 2005.

[WF99] Yuhong Wen and Geoffrey C. Fox. Performance prediction for data intensive applications on large scale parallel systems. Technical report, Northeast Parallel Architecture Center, Syracuse University, 1999.

[WSH00] Richard Wolski, Neil T. Spring, and Jim Hayes. Predicting the CPU availability of time-shared unix systems on the computational grid. *Cluster Computing*, 3(4):293–301, 2000.

[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.

[XZB05] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling Internet Backbone Traffic: Behavior Models and Applications. In *Proc. ACM SIGCOMM*, 2005.

[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.