Efficient Filtering Support for High-Speed Network Intrusion Detection

by

José María González

M.S. (University of California at Berkeley) 2000
Engineering (Technical University of Madrid, Spain) 1995

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor David Wagner, Chair
Dr. Vern Paxson
Professor David Brillinger
Professor Ion Stoica

Fall 2005

The dissertation of José María González is approved:

Chair Date

Date

Date

Date

University of California, Berkeley

Fall 2005

Efficient Filtering Support for High-Speed Network Intrusion Detection

José María González

Abstract

Efficient Filtering Support for High-Speed Network Intrusion Detection

by

José María González

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Wagner, Chair

Network Intrusion Prevention Systems (*NIPS*) are a key element in defending networks against all kinds of malware (worms, virus, etc.). This investigation addresses some functionality and performance issues related to running such systems in very high-speed networks (1 Gbps or faster).

The traditional approach to carry out sound intrusion prevention is the use of software-based approaches, as only they provide the flexibility and dynamic functionality that is required to detect rapidly-evolving malware.

The main obstacle for the deployment of software-based NIPS in high-volume environments is performance, in terms of the amount of traffic the NIPS is able to process. NIPS present a double challenge to system performance, namely processing load and internal state storage management.

We argue that any approach that intends to run NIPS in high-speed links must rely on efficient filtering, i.e., allow the NIPS to decide which traffic it is interested in analyzing and which it is not, in an efficient fashion.

The first contribution of this thesis work is the development of filtering techniques crucial for the operation of network intrusion detection and prevention in high-volume environments. In the first part of the dissertation we discuss new filtering models. We introduce innovative ways to take advantage of traffic filtered using traditional packet filter capabilities, and new mechanisms to extend packet filter capabilities with new fine-grained abstractions.

In the second part of this dissertation, we go a step further with one of the new abstractions discussed earlier, and discuss a packet processing architecture based on implementing the abstraction in a hardware device. The key insight of the approach is that some packet processing tools, including NIPS, can benefit enormously from the addition of a reduced set of very simple operations oriented to performing fast classification of traffic. These operations are simple enough as to permit an extremely fast hardware implementation. We illustrate the performance of the architecture by describing a prototype, and our experience with its usage.

Professor David Wagner
Dissertation Committee Chair

To my parents,

Santos and Maribel

Contents

List of Figures

List of Tables

Acknowledgments

I am very grateful to my advisor, Vern Paxson, for his guidance and support. Working with him has been an unforgettable experience. His help and advice have been determinant in my research, for which I am gratefully indebted.

I would also like to thank the rest of the committee, for their comments and help: Professor David Wagner and Professor Ion Stoica from the Computer Science Deparment, and Professor David Brillinger from the Statistics Deparment (*Muito obrigado pela sua ajuda e generosidade!*).

Special thanks to my former advisor in Berkeley, Professor Larry Rowe. He has been a most inspiring example, research and otherwise. I greatly profited from his involvement in my first few years in Berkeley, and learnt to do research while working with him.

I gratefully acknowledge Nick Weaver, whose collaboration in the Shunting Chapter had been invaluable; Professor Dick Karp, for his precious help in the discussion of associativity influence in hash tables of Chapter 3; Eran Halperin, for the countless number of times he has been willing to provide his help in a large number of subjects; and Eddie Kohler, for his advice in the design of the Shunting environment.

My acknowledge to the German crowd in the Bro group, Holger Dreger, Christian Kreibich, and Robin Sommer, not only for their invaluable help in understanding the details of Bro (in exchange of some *kicker* lessons), but also, and especially, for always smiling when being requested help. *Vielen Dank!*

I am particularly grateful to my advisor in Universidad Politécnica de Madrid, Ángel Álvarez, who encouraged me to continue my studies in the United States.

I am also deeply indebted to the International Computer Science Institute (*ICSI*), where I have worked for the last few years, including two of them under the sponsorship of the Spanish *Ministerio de Educación y Ciencia*. María Eugenia Quintana (*¡Gracias, flaca!*), Lila Finhill, Alberto Amengual, Pedro Ruíz, and all the other people have made my experience at ICSI something to remember forever.

My amazing and astonishing experience in Berkeley would not have been the same without all the people I have befriended, and which have become in some sense my family in the US. Iván Castillo, with whom I had the luck of sharing a house for more than one year, and who ignited my interest and love for Mexico. Guillermo Rein, my best friend in Cal, and with whom I have shared almost everything, including so many experiences, a house for more than three years, and the presence of Nezahualcóyotl. Larissa Muller, who not only agreed to an indigestion of geek lingo by proof-reading this thesis, but also managed to qualify for the roommate category in a quite heterodox way: By her charm, sweetness, ever-lasting capacity to discuss and make people smile, and her endurance to bad puns and worse poetry.

Carlos Arteta, for his friendship and innumerable discussions on politics and life during our first years in Berkeley. I will even forgive his recurring habit of playing Silvio Rodríguez music. Curro Blanch, for so many good hours spent together. Manu Forero and Ryan Schmidt, for demonstrating how far you can reach just with charm.

Sylvia Ratnasamy, my best friend in the Department, the funniest company in so many Nefeli evenings, and the only person that can get a smile out of me at six in the morning after having worked all night long in a dull architecture paper. Eran Halperin and Leticia Ortiz, for all the discussions on life and politics lasting until well past five in the morning, for their confidence, and for always reminding that there is another side for everything. (*Toda raba, achi tov!*) Álex Lago, for his amazing capacity to light up the darkest of the days, and for starting the Spanish Association in Berkeley, where so many good moments have occured. All the members of the Spanish (*Iberia*) and Italian (*IISA*) associations in Berkeley, for creating a social environment in which enjoying the pleasures of life is possible. And a person that should have been here, but is not for a reason we both know.

Celine Monget and Javier Cardona, for always being there, for feeding the hungry and giving drink to the thirsty, for putting up with some of my worst moments, for sharing their *joie de vivre*, and for granting me the utmost privilege of marrying them.

Por último, a mi familia: Santos, Maribel, Santos, Luis, Esperanza, Pablo, María e Ina (que no llegó hasta aquí): si alguna vez leéis esto, os quiero como no se puede querer a nadie. Saber que estáis ahí es el principio y el final de todo.

Chapter 1

Introduction

1.1 Motivation

The popularization of the Internet at the end of last century has produced a tool of immense utility, but also the advent of a multitude of malware. Malware can be defined as software designed specifically to damage or disrupt a system, and it includes such software as viruses, worms, and Trojan horses.

The costs of malware include lost productivity, cleaning up the malware, stolen information, data destruction, and loss in customer confidence. While hard to quantify with exactitude, they are believed to be in the order of several billion dollars per virus.

Malware, a bragging activity in its beginnings, has turned today into a lucrative industry, with more sophisticated and motivated attackers launching both targeted

and generic attacks. With the deeper reliance of advanced societies in the network infrastructure, and the difficulty to track the attackers, the problem is only posed to get worse, and to involve more social and political issues: the Nachi worm caused Air Canada to delay flights by overwhelming its reservation systems, and the Slammer one managed to cripple some machines monitoring the Ohio Davis-Besse nuclear plant.

A key element in defending networks against all kinds of malware is Network Intrusion Prevention Systems (*NIPS*). The basic idea of a NIPS is to monitor all traffic exchanged between the network being defended and the Internet, detect security problems, and block the malware transmissions.

The traditional approach to carry out sound intrusion prevention is to use software-based approaches: Malware is extremely adaptive, and mutates constantly in order to take advantage of new exploits, to add new payloads, to generate polymorphic versions of the same malware, etc. Some malware just resort to go undetected, hiding themselves inside benign traffic. Any sensible detection approach must rely on flexible and dynamic functionality, which only software-based approaches can provide.

Other malware just resort to brute force, trying to infect the largest possible number of victims before defenses can be raised. It is understood that, due to the capacity of worms to spread at very fast speeds, only flexible and automatic defenses are useful against worms [Moore et al., 2003b; Staniford et al., 2002b]. This, again, makes the case for software-based approaches.

The main obstacle for the deployment of software-based NIPS in high-volume

environments is performance, in terms of the amount of traffic the NIPS is able to process. NIPS present a double challenge to system performance, namely processing load and internal state storage management.

The first challenge is processing load: While parsing a packet is in most cases a light task, the amount of traffic in a high-speed link easily exceeds the capacity of the system's processing resources [Dreger et al., 2004].

The second challenge is internal state storage management. In order to soundly analyze network traffic at the network-, transport-, and application-layers, NIPS potentially need to store all the traffic going back and forth between the two connection peers. This may account for a very large amount of data even in a relatively slow connection. How to manage all this information becomes a challenge for the system.

To mitigate both problems, a straightforward approach to permit running NIPS in high-speed links is efficient filtering. The main idea is to permit the NIPS to specify in a fine-grained way the exact subset of traffic it needs to analyze. This reduces the amount of traffic it must deal with, in exchange of skipping traffic that it is not interested in.

This dissertation explores efficient filtering support for NIPS in high-speed networks. It describes new filtering models using traditional packet filter capabilities, and new mechanisms to extend such packet filter capabilities with new, efficient, more fine-grained abstractions.

1.2 Dissertation overview

This dissertation is divided into three chapters and two appendices.

Chapter 2 describes the integration of sampling and filtering into intrusion detection. We propose to augment and enrich the main processing path in stateful Network Intrusion Detection Systems (*NIDS*) by the addition of a parallel, stateless (connection-less) packet-filtering path. We describe several examples on using the new packet-filtering path, and evaluate their advantages, namely simplicity and performance.

Chapter 3 proposes the addition of efficient support for high-speed intrusion detection in packet filters. We propose to extend the widely-used BPF packet filter with two new packet filter mechanisms, namely in-kernel, packet-based random sampling, and in-kernel, fixed-size, generic-purpose, persistent, associative tables. We compare both additions to the current capabilities, and study their benefits.

Chapter 4 justifies and describes Shunting, a novel architecture that permits high-speed, extensive (non-sampled and in-depth), stateful, inline traffic processing by integrating a simple, active, hardware device with a complex, software, decision engine. We present a prototype implementation of the Shunting architecture, and the modification of a popular NIDS in order to serve as its engine. We evaluate its performance, and suggest other fields different from intrusion detection where it can be used.

Chapter 2

Integration of Sampling and Filtering into Intrusion Detection

2.1 Abstract

This Chapter describes the integration of sampling and filtering into network intrusion detection. We propose to augment and enrich the main processing path in stateful Network Intrusion Detection Systems (NIDS) by the addition of a "Secondary Path," a parallel, stateless (connection-less) packet-filtering path.

In a stateful NIDS, the main packet-capture path (Main Path) performs network- and transport-layer analysis, and provides a framework for application contents analysis. It receives the raw traffic and analyzes the lower communication layers. After this analysis, the application contents are dispatched to the corresponding application-

layer analyzer, which performs its specific analysis.

The Secondary Path is an alternate channel for acquiring packets. It provides a network-layer framework for traffic analysis. In other words, the traffic is served directly to the analyzers, without any previous analysis.

The main benefit of using the Secondary Path is that analyzers that use it may take advantage of flexible filtering and sampling, while at the same time avoiding the cost associated to the Main Path performing full connection-oriented analysis. While a simple addition, we claim that, in some scenarios, this alternate traffic processing can provide useful information that can complement or disambiguate the information obtained by the Main Path.

We introduce the Secondary Path, justify it, and present an implementation on a popular stateful NIDS. We also show several examples of its use.

2.2 Introduction

Monitoring network traffic from a security perspective is required to manage today's networks. Malicious activity, from portscanning or denial of service attacks to viruses and worms, is a continuous presence in communication networks, and presents serious challenges to their day-to-day operation.

In order to prevent or detect the presence of malicious activity, one of the main tools available are Network Intrusion Detection Systems (*NIDS*). NIDS are systems that detect malicious network activity by monitoring network traffic [Mukherjee et

al., 1994].

NIDS work by capturing packets from the network and analyzing them. Different analyzers check the correctness of the various protocol layers, and produce events when observing anomalies. While not all NIDS do actually parse all the different protocol layers, it is well understood in the research community that only full protocol analysis, from the network layer up to the application layer, provides a sensible defense against attacks to operative networks. This is known in the literature as "deep packet analysis."

Moreover, NIDS analysis must be stateful. Sound application-layer protocol analysis may require access to the full application-layer contents (the "Application-layer Data Unit," or *ADU*). ADU contents may be spread along different packets, and therefore packets can be processed only by considering them in the context of their connection.

When receiving a new packet, a NIDS must consider it in the context of existing information on the packet's connection. This context is obtained from already-received packets from the same connection, which the NIDS must have stored. Connection-oriented dependencies not only extend to the past, but also to the future: The NIDS may not be able to complete the processing of the packet until it receives further traffic from the connection.

A stateful NIDS may therefore need to store an indefinite amount of per-connection data for an indefinite amount of time. This data includes ADU contents, per the nor-

mal packetization issues just discussed. It may also include network- and transport-layer contents, in order to be resilient to attacks based on ambiguities [Ptacek and Newsham, 1998].

Deep, stateful per-packet monitoring of a high-speed link to detect security intrusions is a resource-intensive task. Each packet must be captured and analyzed, and in some cases, stored. The analysis part, i.e., deciding whether a packet poses a security threat or not, may require a considerably complex processing effort. The storage part may require a considerably expensive bus and memory access effort. Operational use in a high-volume environment intensifies the problem by increasing the amount of traffic that must be processed.

This problem is magnified by two other effects, namely traffic diversity and state management. First, as the amount of traffic increases, the traffic diversity and the crud in the link also increase, which produces not only more false alarms, but also more diverse ones [Dreger et al., 2004].

Second, the amount of state needed to produce a good snapshot of the network state in stateful NIDS grows with the amount of traffic processed. This creates an enormous state management problem.

This Chapter proposes the use of two packet-capture paths with different services in the context of network intrusion. While a NIDS traditional path (deep and stateful) is a must for some analyzers, others may be willing to tradeoff isolated-packet processing in exchange of efficiency. The latter is achieved by reducing the amount

of traffic received using filtering and/or sampling.

Our idea is developed in the "Secondary Path," a lightweight, stateless, packet-capture path that complements a NIDS traditional, deep, stateful packet-capture path (which we name the "Main Path"). We describe an implementation of the Secondary Path on Bro [Paxson, 1999], a popular, stateful, open-source NIDS. We also present several applications that use it, including large connection, backdoor, and P2P traffic detection.

To our knowledge, this is the first time that packet paths with differentiated services have been proposed in the context of intrusion detection. We are not aware of any NIDS that combines a stateful path with a stateless one.

The rest of the chapter is organized as follows: Section 2.3 introduces related work. Section 2.4 presents the Secondary Path, an implementation, its operation, and how to use it. Section 2.5 discusses several applications of the Secondary Path in a stateful NIDS. Section 2.6 concludes.

2.3 Related Work

The goal of Network Intrusion Detection Systems is to detect attacks on computers, especially those carried out over the network. Section 4.3.1 in Chapter 4 discusses related work on NIDS.

The Chapter describes the integration of sampling and filtering into network intrusion detection. The main basis of filtering is Packet Filters. A *Packet Filter* is

a mechanism to select packets from a packet stream using a programmable criterion (the filter). Related work on Packet Filter models is presented in Section 3.3.1 in Chapter 3.

The other integration idea is sampling. Section 3.3.3 in Chapter 3 discusses related work in sampling.

The research work related to the particular applications implemented on the Secondary Path is described in the context of Section applications

Work related to the particular applications being implemented on the Secondary Path is described in the context of the applications themselves.

2.4 Secondary Path

2.4.1 Main Path

The operation of a typical stateful NIDS consists of (a) capturing traffic from one or several packet-capture devices, (b) checking network- and transport-layer contents, (c) reassembling the application-layer contents, and (d) handing them out to the corresponding analyzer. We call this mechanism to process traffic the "Main Path."

The Main Path provides a framework for application-layer traffic analysis: It is used by analyzers that perform connection-oriented, ADU analysis.

The connection-oriented nature of the Main Path permits hiding the details of the reassembling from the application-layer analyzers. The Main Path reassembles the

application-layer payloads of different packets, and dispatches them to the analyzers. The latter are provided with full application-layer payloads for deep analysis, plus some connection information.

The main drawback of providing full application-layer analysis is that the traffic processed by the Main Path must be composed of full connections. This limits substantially the usage of input-volume control techniques (sampling or filtering), which are needed for performance reasons.

For example, filtering can be based solely in the five fields that compose the connection tuple (source and destination address and port, plus transport-layer protocol). A well-known, efficient operation mode for NIDS consists of limiting the amount of traffic they must process by focusing in just a subset of the protocols (port-based filtering), instead of parsing all the traffic in the wire.

Sampling can only be connection-based, which is not available in current packet-filters and has different properties than packet-based sampling. [1]

We believe that, while full-payload analysis is required for sensible deep, stateful analysis, there are some cases where complementary information can be obtained more efficiently from analysis of isolated packets. The information obtained in such a way is independent of the obtained from the Main Path, and can be used to complement or disambiguate the latter.

[1] Chapter 3 of this thesis discusses connection- based sampling in the context of a popular packet-filter, BPF.

2.4.2 Secondary Path Description

The Secondary Path is an alternate channel for acquiring packets. It works by capturing packets from one or several packet-capture devices, and handing them out to the corresponding analyzer, without any previous analysis.

It is very important to remark that the Secondary Path is an alternate channel: It provides a stateful NIDS with a means to obtain information about the monitored traffic whose generation using the Main Path is either inefficient or ambiguous. It does not substitute the Main Path. Instead, it complements it.

The Secondary Path is simpler than the Main Path: Analyzers are served with isolated packets, instead of full connections. No reassembling is carried out, and therefore no state must be kept. A packet is received, dispatched to the interested analyzers, and then discarded.

The consequence of not performing packet reassemble are that analysis through the Secondary Path is susceptible to evasion. In other words, it is easy for an attacker to avoid a Secondary Path analyzer by fragmenting her traffic adequately [Ptacek and Newsham, 1998].

Note, however, that while the Secondary Path is stateless *per se*, analyzers that use it may be stateful, by relying in the NIDS state capabilities.

The Secondary Path provides a framework for network-layer packet analysis. Analyzers receive network-layer headers and payloads, which are the only ones that are guaranteed sensible in the absence of connection context.

Analyzers based on the Secondary Path must be careful when using transport- or application-layer contents. Both may be divided among several packets, which may arrive to the destination out of order, or even duplicated.

Figure 2.1 compares the Main and Secondary Path. The Main Path receives traffic, and performs network- and transport-layer analysis, and hands processed contents (ADUs) to the analyzers. The Secondary Path just dispatches packets to the analyzers. Its typical use is the monitoring of low-bandwidth, connection-less packet subsets.

The main advantage of the Secondary Path is efficiency. While the same information that can be obtained with the Secondary Path can also be obtained with the Main Filter by analyzing the whole traffic, the latter performs network- and transport-layer analysis. We present some examples where this analysis is unneeded.

Filtering

The other main advantage of the Secondary Path is that it allows analyzers to make extensive use of filtering and/or sampling. As a consequence, it diminishes the amount of traffic the NIDS must process, which helps it to operate in high-speed environments.

It is often possible to extract useful information from a packet stream by analyzing a small subset of the traffic. This subset can be defined by specifying a static, simple packet-filter *expression*. This expression may be based in network- and/or transport-

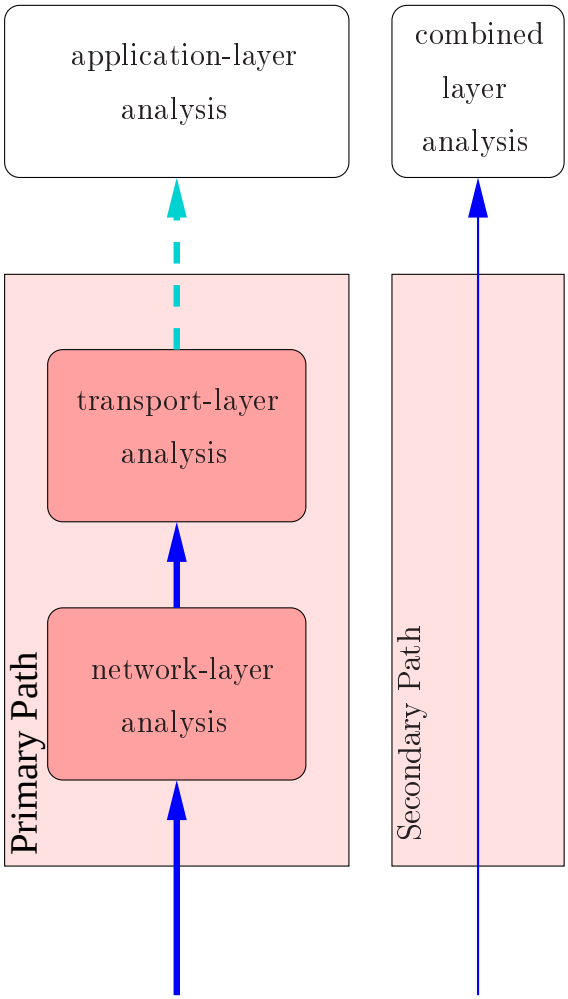


Figure 2.1: Main vs. Secondary Path

layer headers, which are easily accessible using standard expressions. Note that the transport-layer based filtering case is less reliable, as TCP or UDP contents may be divided among several IP packets. On the other hand, if there is no evasion, this is relatively uncommon and, what is more, typically restricted to a small subset of protocols [Shannon et al., 2002].

Filtering may also include application-layer contents. While packet-based filtering is limited to matching bytes located in fixed packet positions, modern application-layer protocols sometimes use headers with distinctive contents in fixed positions [Zhang and Paxson, 2000a].

For example, HTTP request headers start with one of 7 different method string ("GET", "POST", etc.), and HTTP response headers start always with the string "HTTP/" [Fielding et al., 1999]. In the same way, SSH connections always transmit the SSH client and server version by sending a packet with the strings "SSH−1" or "SSH−2" [Ylonen, 1996].

An analyzer that receives packets from the Secondary Path, and that uses as filter that the first 5 bytes of the packet payload are "HTTP/", access one and only one packet per HTTP connection, with high probability. With persistent HTTP connections, the analyzer will likely receive one packet per entity, as entity headers are typically sent in a different packet than the previous entity body. The analyzer will also be able to access to HTTP responses in non-standard ports.

Creating a packet-filter expression that captures SSH connections or HTTP re-

quests is just slightly more complicated.

Due to the fixed-location limitation of packet filtering, and the stateless condition of the Secondary Path, application-layer contents provide less leverage than network- or transport-layer contents.

For example, if she wants to avoid an HTTP connection being detected, she may just fragment the first 5 bytes in two different packets. If she wants to increase the traffic received by a NIDS trying to detect HTTP traffic, she may just forge faked packets starting with the mentioned 5 bytes.

Sampling

The second mechanism that permits thinning the amount of traffic analyzed by the NIDS is packet-based sampling. Packet-based sampling generates a completely unstructured traffic stream, whose main properties are related to those of the original stream [Duffield et al., 2002, 2003].

A case example of sampling is identifying heavy hitters, i.e., connections or hosts that account for large subsets of all the traffic. If the sampling is unbiased, a heavy hitter in the total traffic is very likely a heavy hitter in the sampled traffic.

Drawbacks

The first drawback of the Secondary Path is that is susceptible to evasion. It is easy for an attacker to avoid a Secondary Path analyzer by fragmenting her traffic

adequately [Ptacek and Newsham, 1998].

The Secondary Path is not intended for analyzers that require full application-layer content access. While the analyzer may write a filter that provides access to the full connection, and then reassemble the contents itself, this is very inefficient, and better suited for the Main Path.

Last, filtering always implies a tradeoff, as information that could be used to refine a conclusion has often been filtered out. Moreover, an attacker knowledge of the filter can be used to either evade or overwhelm the detector.

Table 2.1 summarizes the main differences between the Main Path and the Secondary Path.

	Main Path	*Secondary Path*
processing offered	L7 analysis	L3, and some L4 and L7 analysis
processing carried out	L3, L4 analysis	none
L4 reassemble	yes	no
memory	stateful	stateless
filtering flexibility	poor (port-oriented)	richer when coupled with stateful BPF (see Section 3.7)
sampling	static, connection-oriented only	richer when coupled with randomness in BPF (see Section 3.4)

Table 2.1: List of Differences between the Main and Secondary Paths

New Filtering Models

We think there are enough opportunities for leverage sampling and filtering using the Secondary Path. Among other, we expect the Secondary Path to be useful for

analyzers that can make do with a small, easily-definable subset of the traffic, namely:

- pseudo-random sampling. For example, the heavy hitters detector described in Section 2.5.3 uses a random-sampling filter, for example "random(1000)".
- deterministic sampling. Section 2.5.2 introduces a "large connection detector" based on a filter whose functionality is "tcp.seq inside a series fixed ranges".
- network- and transport-layer headers. Section 2.5.4 discusses a generic backdoor detector based on TCP header contents.
- application-layer content signatures easily expressible with BPF. Section 2.5.4 presents a payload-based backdoor detector.

2.4.3 Operation

The operation of the Secondary Path is fairly simple: Analyzers provide a packet-filter expression that defines the traffic subset for which they are interested in performing isolated packet analysis.

The Secondary Path creates a filter resulting from the union of all the analyzer filters (Secondary Filter), and opens a packet-filter device with it.

When a packet matches the common filter, the Secondary Path runs each particular analyzer filter against the packet, and dispatches the latter to those analyzers whose filter matches the packet.

Note that, during the Secondary Path operation, each analyzer filter is actually run twice: first as a part of the full Secondary Filter, and second as the analyzer's particular filter. This does not present problems with the BPF packet filter, as BPF filters are idempotent: running a filter F over a set of packets already filtered by F does not cause the rejection of any packet.

On the other hand, when adding state or randomness to BPF (see Chapter 3), filters are not anymore idempotent, and the Secondary Path's double filtering may not produce the expected results for analyzers using filters based in pseudo-random sampling. A quick solution is to identify those filters using pseudo-random filtering, and set a separate packet-filter device for each of them.

2.4.4 Implementation

We have implemented the Secondary Path in a stateful NIDS, namely Bro [Paxson, 1999].

The implementation of the Secondary Path is fairly simple: Analyzers associate to the Secondary Path a tuple formed by a packet filter expression, and a Bro event. In order to do so, the application adds an item to the global table secondary_filters, associating a string index (the packet filter expression) with an event yield (the event that will be raised when a packet matches the expression).

The interface(s) being monitored is open twice, one for the Main Path, and another for the Secondary Path. The expression used for the Secondary Path packet filter is

the OR'ed juxtaposition of all client redefinitions to the secondary_filters table. In particular, the Main and Secondary Path expressions are independent, and each opens its own packet-filter device.

Whenever a packet matches the Secondary Path, every filter associated to it is run against the packet. For those filters that match the packet, the corresponding event is fired with the packet as one of the arguments.

Figure 2.2 shows how to use the Secondary Path. The code will cause Bro to invoke the event SFR_flag_event for every packet that matches the expression "tcp[13] & 7 <> 0", i.e., every time there is a TCP packet with any of the SYN, FIN, or RST flags set.

Note that the event interface provides only the network- and transport- headers, but not the application-layer payloads. The reason is that Bro's script language makes access to binary contents clumsy, and so far, none of the detectors we have written has needed the application-layer contents.

In the current implementation, analyzers are provided only with network- and transport-layer headers of packets matching their filters. While we could make the payload available, this presents some implementation problems, as the Bro script language is not well-suited to support binary payload access. Moreover, we have not yet seen any case where access to the application-layer contents is required to do the analysis.

```
redef secondary_filters += { ["tcp[13] & 7 != 0"] = SFR_flag_event}

type tcp_hdr: record {
  sport: port;     # source port
  dport: port;     # destination port
  seq: count;    # sequence number
  ack: count;    # acknowledgment number
  hl: count;     # header length (in bytes)
  dl: count;     # data length (xxx: not in original tcphdr!)
  flags: count;    # flags
  win: count;    # window
};

type udp_hdr: record {
  sport: port;     # source port
  dport: port;     # destination port
  ulen: count;     # udp length
};

type icmp_hdr: record {
  icmp_type: count; # type of message
};


type pkt_hdr: record {
  ip: ip_hdr;
  tcp: tcp_hdr &optional;
  udp: udp_hdr &optional;
  icmp: icmp_hdr &optional;
};

event SFR_flag_event_event(filter: string, pkt: pkt_hdr)
  {
  }
```

Figure 2.2: Secondary Path Use Example

2.4.5 Example

An example of the usage of the Secondary Path to obtain complementary information is backdoor detection (see Section 2.5.4). [Zhang and Paxson, 2000a] suggests several

mechanisms to detect interactivity in connections. These mechanisms do not require full analysis of each connection. For example, in the "Generic Algorithm for Detecting Interactive Backdoors," only the timing and the frequency of the small packets is required to detect interactive traffic. On the other hand, the analysis cannot be limited to just a subset of the ports: A backdoor, by definition, normally runs on a non-standard port, so that it can hide itself from security monitoring.

An example of the usage of the Secondary Path to disambiguate information obtained in the Main Path is large connection detection (see Section 2.5.2). A common, cheap approach to monitor TCP connection sizes consists of subtracting the TCP sequence number field at the end of the connection from the same field at the beginning of the connection. Unfortunately, this mechanism produces completely wrong results with extremely large connections that wrap up the sequence number space, or when dealing with broken TCP implementations.

2.4.6 Performance

Experiments Configuration

Unless otherwise noted, all experiments described in this Chapter have been carried out in a single-processor, Intel Xeon (Pentium) CPU running at at 3.4 GHz, with 512 KB cache and 2 GB of total memory. The host operating system was FreeBSD 4.10.

All times reported are the addition of the user and system times, as reported by

the Operating System. All experiments have been run in an idle host.

Experiments were run 100 times, and the standard deviation calculated. In all cases the standard deviation was negligible compared to the average times.

Empty Event Performance

The first interesting feature of the Secondary Path is its cost. In order to measure this cost, we have used the Secondary Filter to run an empty event, i.e., an event that does not carry out any work, and returns as soon as it is raised.

This cost depends not only on the number of packets that raise the Secondary Path event, but also on the number of packets than do not raise the Secondary Path event, but must be read by the kernel and eventually discarded by the Secondary Filter.

Figure 2.3 shows the performance of the Secondary Path using an empty event. Note that both scales are logarithmic.

The thick line represents the cost of rejecting packets with the Secondary Filter. It was obtained by running the Secondary Path with just one detector whose filter matches no packets. (The experiment was run for several traces of different sizes.)

We call this cost "fixed", as it is independent of the number of packets accepted by the Secondary Filter. It is the sum of two effects, namely (a) the fixed cost of running Bro, and (b) the cost of accessing all the packets in the stream and filter them (even if they are all rejected). It is clear that the first effect is more important

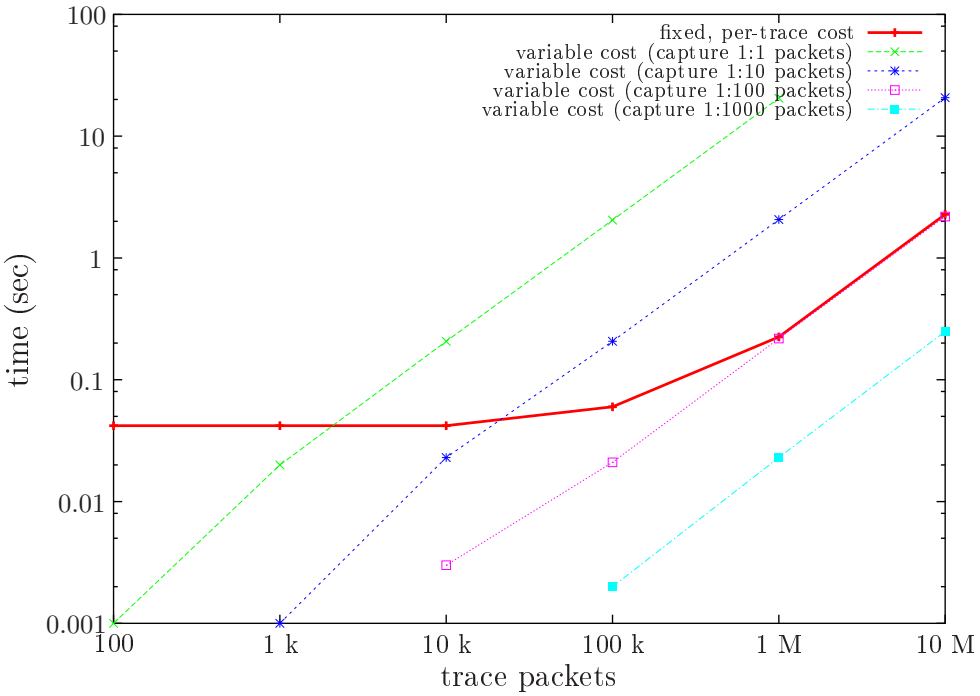


Figure 2.3: Performance of the Secondary Path with an Empty Event

with small traces (this is the flat part to the left of the 10 K packet mark), while the second effect is more important with large traces (cost increase to the right of the 1 M packet mark).

The dashed and dotted lines show the cost of an empty detector (a detector whose event returns immediately), when a given ratio of the packets match the filter. We call this cost "variable", as it depends on the ratio of packets reaching the Secondary Filter event. We have subtracted the fixed per-trace cost in order to separate the fixed and variable (per-matching packet) costs.

There are two interesting insights in Figure 2.3.

First, the variable cost is proportional to the ratio of packets matching the filter. In other words, the variable cost of sampling, say, 1 in 10 packets is 10 times larger

than the variable cost of sampling 1 in 100 packets.

Second, the fixed cost of running the Secondary Path with a very simple event is similar to the variable cost of capturing 1 in 100 packets. This means that, if the Secondary Path event processing is simple, whether the detector's filter matches 1 in 1000 packets or 1 in 10000 packets does not affect the Secondary Path overhead. It is when the ratio gets close to 1 in 100 packets that the Secondary Path cost starts being affected by the ratio of captured packets.

2.5 Applications

We have created 3 examples of tools that take advantage of the Secondary Path technique to augment the reach of the Main Path.

Section 2.5.1 discusses the trace used for most application experiments.

Section 2.5.2 introduces a Large Connection Detector. The goal of this detector is to disambiguate large connection information. It uses a low-bandwidth, deterministic filter (a series of equidistant stripes in the TCP sequence number range).

Section 2.5.3 describes a Heavy Hitters Detector. The goal of this detector is to discover heavy traffic patterns. It uses a low-bandwidth, pseudo-random sampling filter.

Section 2.5.4 presents an implementation of the Backdoor Detection algorithms presented by [Zhang and Paxson, 2000a]. The goal is to detect interactive traffic in non-standard ports, which is often associated to backdoors. It uses a series of

low-bandwidth, deterministic filters.

2.5.1 Trace

The trace used for most experiments (named *tcp-1*) was obtained at the Lawrence Berkeley National Laboratory (LBL) DMZ, whose link is 1 Gbps. It consists of TCP-traffic only, and accounts for 1.2 M connections, 127 M packets, and 113 GB (an average of 892 bytes/packet). The trace was taken on a weekday's working hour, in September 2005. Its total duration is 2 hours (an average bitrate of 126 Mbps).

2.5.2 Large Connection Detection

Rationale

The first example of filtering is a large TCP connection detector. This is an example of static traffic characterization.

Related Work

The goal of traffic characterization is to summarize the traffic in a link by describing the quantitative importance of several categories. These categories may be defined by using one or several criteria alongside multiple dimensions.

An application of traffic characterization is getting the largest connections in a link. The criterion used to categorize the traffic is the traditional 5-tuple (104-bit) connection definition.

A related application is traffic matrices, in which the categories are each of the combination of 2 nodes exchanging traffic [Medina et al., 2002]. Traffic matrices are used for several purposes, including designing network topologies, planning link capacities, and configuring network routing policies. Note that "nodes" may be IP addresses or subnetworks.

Note that both applications are examples of static traffic characterization: the exact criterion used to categorize the traffic is defined *a priori*. The analysis is relatively simple: Every time there is a new packet, its corresponding categories are identified and their size updated.

The main issue in static traffic characterization is performance: In high-speed links, the amount of packets that need be processed may be large enough as to limit the per-packet processing budget to just a few operations per packet (processing concern). At the same time, the number of connections may be large enough as to preclude keeping information about each connection in fast, small memory (memory concern).

The traditional research in static traffic characterization consists of proposing mechanisms that permit spending the reduced resource budget only in the large categories, which are the ones being measured anyway.

For example, [Mitzenmacher and Upfal, 2005] proposes the use of *count-min* filters to count large connections. Count-min filters work by setting a 2-dimensional array of counters (k groups of m/k counters each, totaling m counters), whose initial values

are zero, and which should be big enough as to fit the total amount of traffic being measured.

When a packet arrives, its connection identifier (the 5-tuple composed by the source and destination address and port, plus the transport-layer protocol) is hashed using k Universal Hash Functions ("UHF"). UHFs select one counter per group, which is incremented by the packet size.

In order to calculate the size of a connection, its connection identifier is hashed again using the k UHFs. From the k selected counters, the one with the smallest counter is selected as the size of the connection.

The authors show that the smallest counter associated with a connection is an upper bound in its real size, and with bounded probability, it is off by no more than ϵ times the total number of packets processed, where ϵ can be obtained by solving Equation 2.1

$$\left(\frac{k}{m\epsilon}\right)^k = e^{-m\epsilon/e} \tag{2.1}$$

The authors also propose the use of *conservative update*, where from the k counters selected by the UHFs, only the one with the smallest count is always updated. For the remaining ones, it is ensured that no counter ends up with a smaller value than the previously-updated smallest count.

[Estan and Varghese, 2002] proposes the use of "sample and hold" and multistage filters to efficiently estimate statistics of large flows. "Sample and hold" is a technique to measure the traffic incurred by the largest traffic flows in high-speed environments by using a relatively-small SRAM cache (the "connection cache"). The idea consists of storing per-connection information in the connection cache. Packets received are only accounted for if their connection is already in the connection cache, or if they are randomly chosen to occupy a new entry in the connection cache.

Multistage Filters has the same large-connection accounting goal. In this case, the idea is to hash the connection tuple, and when the corresponding entry exceeds a given threshold, add its information to the flow cache. Multistage Filters are prone to false positives, as several small flows may hash to the same entry. In order to avoid them, the authors propose to use several stages using independent hash functions.

Large-Connections Detector Description

A cheap mechanism that is often used to calculate the amount of traffic in a stateful (TCP) connection consists of comparing the sequence numbers at the beginning and at the end of the connection, and subtract them. Unfortunately, this is not very reliable in (a) connections that do not terminate or for which the NIDS misses their establishment, (b) very large (greater than 4 GB) connections that end up wrapping around the TCP sequence number (note that this is allowed in TCP while there is no ambiguity on what a packet's sequence number means, due to its use of a window

smaller than 2 GB in size), and (c) broken TCP stacks that cause incorrect sequence numbers, especially at the RST segment.

Operation

The large connection detector works by filtering for several thin, equidistant, randomly-located stripes in the sequence number space. A truly large flow will pass through the stripes in an orderly fashion, maybe several times. The detector will keep track of all packets that pass through any of the stripes, counting the number of times a packet from a given flow passes through consecutive regions (K).

For example, if we lay down 4 stripes separated 1 GB in the 4 GB-long TCP sequence number range, and we see a connection passing through 2 consecutive stripes ($K = 1$), we know that the connection has likely accounted for at least 1 GB.

It is important that the first stripe is located randomly, i.e., that the location field of the detector mask is chosen randomly. This way, an attacker cannot predict which sections of the sequence space are being monitored. Thus, she cannot overwhelm the detector by sending lots of packets in the stripes.

Note that the detector returns always two guesses, an upper limit on the amount of traffic, and a lower limit. If a connection has been seen in two consecutive stripes, the estimated size may be as large as the distance between 4 consecutive stripes, or as small as the distance between 2 consecutive stripes. In the previous example, we know that the connection has likely accounts for at most 3 GB of traffic.

Figure 2.4 shows an example with 4 stripes. The 4 horizontal stripes, named s_A, s_B, s_C, and s_D, respectively, represent the parts of the TCP sequence number range where the detector is "listening" for packets. The thick diagonal lines depict the time and TCP sequence number of the packets of a given TCP connection. The dotted, vertical lines represent events in the Secondary Path. Note that we could use a different number of lines, and lines with different width. This is discussed in the next Section.

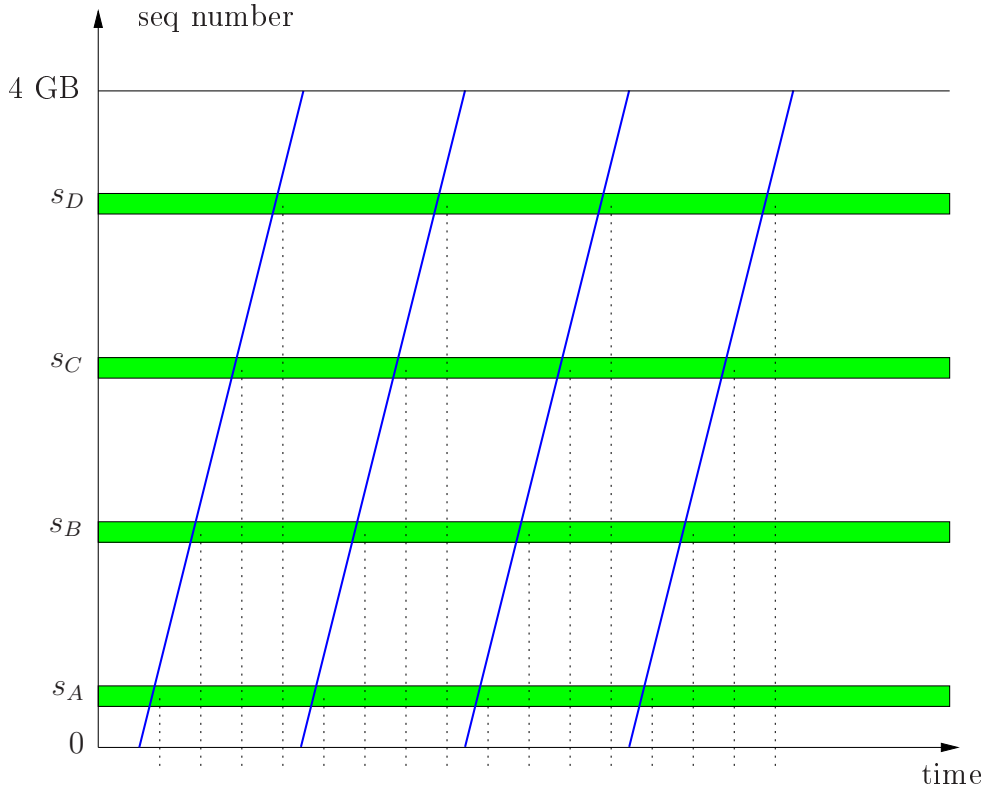


Figure 2.4: Large Connection Detector Example

The first stripe is located randomly in the sequence space. The remaining ones are located at a fixed distance from the first, which divides the TCP sequence number range in equidistant zones.

For the detector implementation, the main difficulty is to calculate the tcpdump expression that will identify TCP packets falling in any of the stripes. This expression is always of the form "seq & mask == value", where seq is the packet's sequence number, mask is a configuration mask, and value is fixed.

Figure 2.5 shows the structure of a generic large connection detector expression.

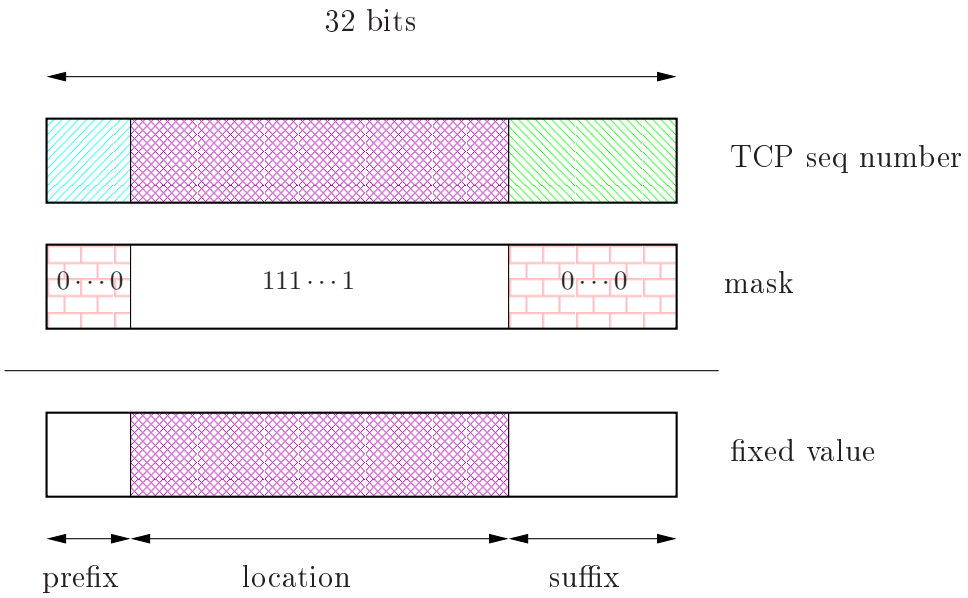


Figure 2.5: Large Connection Detector Expression

The expression is calculated by dividing the 32 bit TCP sequence number field in three parts, a *prefix*, a *location*, and a *suffix*.

The *prefix* represents the number of stripes the detector is using. Its length is the logarithm in base 2 of the number of stripes. In our example, therefore, its length would be 2 bits. As the detector must capture packets whose sequence number is located in any of the stripes, the expression must accept any value in the corresponding TCP sequence number bits. The corresponding bits in the mask are

therefore reset. Note that by locating the stripe index as a prefix, we manage to lay the stripes in an equidistant fashion.

The *suffix* represents the stripe size, and its size is the logarithm in base 2 of the stripe size (in bytes). In the example, its length would be $\log_2(2048) = 11$ bits. As the detector must capture packets whose TCP sequence number falls in any place of a given stripe, the expression must accept any value in the corresponding TCP sequence number bits. The corresponding bits in the mask are therefore reset.

The *location* field states the exact location of the stripes. It occupies the remaining bits between the prefix and the suffix. The detector must only capture packets whose TCP sequence number falls in a given set of stripes, i.e., whose location field has a fixed value. Therefore, the corresponding bits in the mask are set to one. In the example, the bits not used for prefix or suffix (19) are used to set the location. In this case, the final mask will be 00 11111111111111111111 00000000000, or 0x3FFFF800 in hexadecimal.

Let's assume we choose as the location field of the fixed value location the binary number 01010101010101010. Therefore, the final fixed value will be will be 00 01010101010101010 00000000000, or 0x15555000 in hexadecimal. The final tcpdump expression will be "(tcp[4:4] & 0x3FFFF800) == 0x15555000".

The connection depicted in Figure 2.4 is a truly large connection, wrapping around the TCP sequence range 4 times. It is therefore seen 16 times by the detector. The size will be estimated in 16 GB.

Traffic Incoherences

An important case on the detector operation is the existence of incoherences.

We define a *transition* as the capture of two consecutive packets of the same connection in different stripes[2]. For example, if two consecutive packets for a given connection are seen in stripes X and Y, where $X \neq Y$, we say that the detector saw a transition (X, Y) for the connection.

We say a transition (X, Y) is *valid* when $Y = X + 1$. In other words, when the two captured packets fall in two consecutive stripes. Otherwise, the transition is said to be *invalid*, and the connection is said to have an incoherence. Incoherences are due to bogus connections, but also to network and stack effects such as packet reordering, losses, or retransmissions.

For every new packet captured in a stripe, the detector's current operation follows three steps: First, if the transition is valid, it adds one to the connection's K counter. Second, if the transition is invalid, it adds one to the connection's invalid transition counter. Third, the stripe identifier of the last packet is recorded.

As an example, seeing the connection consecutively in stripes 7, 8, 9, 10, and 11 implies the detector counts 4 valid transitions, namely $(7, 8)$, $(8, 9)$, $(9, 10)$, and $(10, 11)$. After all the transitions, $K = 4$.

The main advantages of this approach is that it is cheap, simple to implement, and rejects bogus connections. Tracking a connection requires only three counters of

[2]When two consecutive packets of the same connection are seen in the same stripe, the detector just ignores the second one.

detector state, namely the identifier of the last stripe in which the connection was seen, K, and the number of incoherences caused by the connection. Processing a packet is a simple three-step task. Bogus connections will appear in few or no stripes, and will only increase their number of incoherences.

The main drawback is that incoherences caused by pathologies in the network (retransmissions or reordering) will cause the detector to underestimate the real size of the connection.

Consider a connection where packet retransmissions occur. Assume the detector sees the connection consecutively in stripes 7, 8, 7, 9, 10, and 11. In this case, transitions $(8, 7)$, and $(7, 9)$ are invalid, and the connection's K is not incremented. The remaining transitions are valid, and therefore the final value of K will be 3.

Consider a connection where packet reordering occurs. Assume the detector sees the connection consecutively in stripes 7, 9, 8, 10, and 11. In this case, transitions $(7, 9)$, $(9, 8)$, and $(8, 10)$ are invalid, and the corresponding K is not incremented for them. The final value of K will therefore be 1 (corresponding to the only valid transition, $(10, 11)$).

Note that incoherences caused by network pathologies are much more likely to occur when the stripes are closer, i.e., when the number of stripes is large.

Detector Tuning

The large connection detector can be tuned by setting the number of stripes or the stripe width.

First, the detector permits increasing the definition of the returned information, by increasing the number of stripes. This trades off processing, as more packets will be filtered in, in exchange of definition.

We know that the TCP sequence range is 4 GB. If the number of stripes is S, we know that the distance between stripes is $D = 4$ GB / S. The size of a connection seen in K consecutive stripes will be reported as $(K-1)D < \text{size} < (K+1)D$. It follows that the absolute error for either the upper limit or the lower limit will never be larger than than $\epsilon = 2D$. By operating in the middle of both limits, the maximum error of the size estimate is ϵ / $2 = D$

For example, in the case of 4 stripes, we know $D = 1$ GB, and a connection seen in two consecutive stripes will be larger than 1 GB, but smaller than 3 GB. By using 2 GB as the average value, we would never be off by more than 1 GB.

The second tuning parameter is the stripe size, W. Assuming the Secondary Path produces no packet drops, the stripes need be wide enough to ensure that, in a sequence of maximum-size packets, at least one of them is captured by the filter. This means stripes need only be as large as the maximum network packet size, which is usually 1500 bytes (or 2 KB to make for an easier operation). [3]

[3] The appearance of *jumbo frames* (9 KB) in current networks is very small [Dykstra, 1999].

When the Secondary Path suffers packet drops, the detector may not see the passing of a connection through a stripe, and wrongly assume that the connection is misbehaving. Wider stripes ensure that packets drops will not affect the reliability of the detector, and therefore make the detector more accurate. The cost of using wider stripes is, again, that more packets will be filtered in, and therefore more processing will be needed.

Let's show an example: Assuming the Secondary Path sees no packet drops, the stripes need only be $W = 2$ KB wide. If we want to detect large TCP flows with a maximum error of $\epsilon = 4$ MB, we know that $D = 2$ MB, and therefore $S = 4$ GB / $D = 2$ K stripes. The total space being monitored in the TCP sequence range will be $SW = 4$ MB, which, considering the full range (4 GB), means that an average of 1 in 1024 packets will be captured.

Results

We ran the Large Connection Detector in the trace described in Section 2.5.1. We tried several values for the number (S) and the width (W) of the stripes.

Figure 2.6 shows, for the largest connection in the trace (3.5 GB application-layer payload), its real size, the upper and lower estimations reported by the detector, and the average of the last two (the *average estimation*), for different number of stripes, and for 2 KB-wide stripes. The stripe width does not affect significantly the correctness of the detector, which makes sense as the experiment was done off-line,

and therefore there were no packet drops.

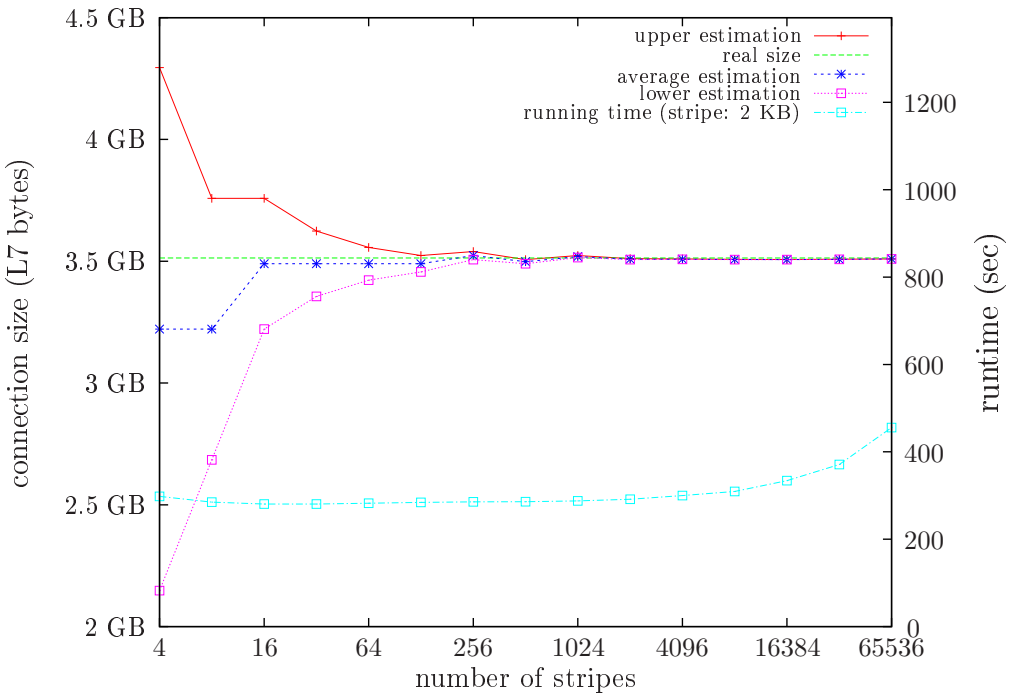


Figure 2.6: Detector Estimation for a Large Connection

In order to show the overall performance of the detector, and the influence of network pathologies, Figure 2.7 shows, for the largest 50 connections in *tcp-1*, the average of the detector's *average estimation* absolute error (thick line), and compares this error with the average number of incoherences for the same set of connections (dashed line). We use absolute instead of relative errors because the detector's error depends on the number of stripes S, but not on the size of the connections.

The left side of Figure 2.7 (up to 256 stripes) shows a significant decrease in the average absolute error. There are almost no incoherences reported, and reported errors are both positive (overestimation) and negative (underestimation).

The right side of Figure 2.7 (more than 2K stripes) shows a stabilization of the

absolute error around 6 MB, and a significant increase in the number of incoherences. Almost all errors are negative. As expected, closer stripes implies a higher probability that the detector gets confused by network pathologies (increasing number of incoherences), which causes systematic underestimation of the connection's real size.

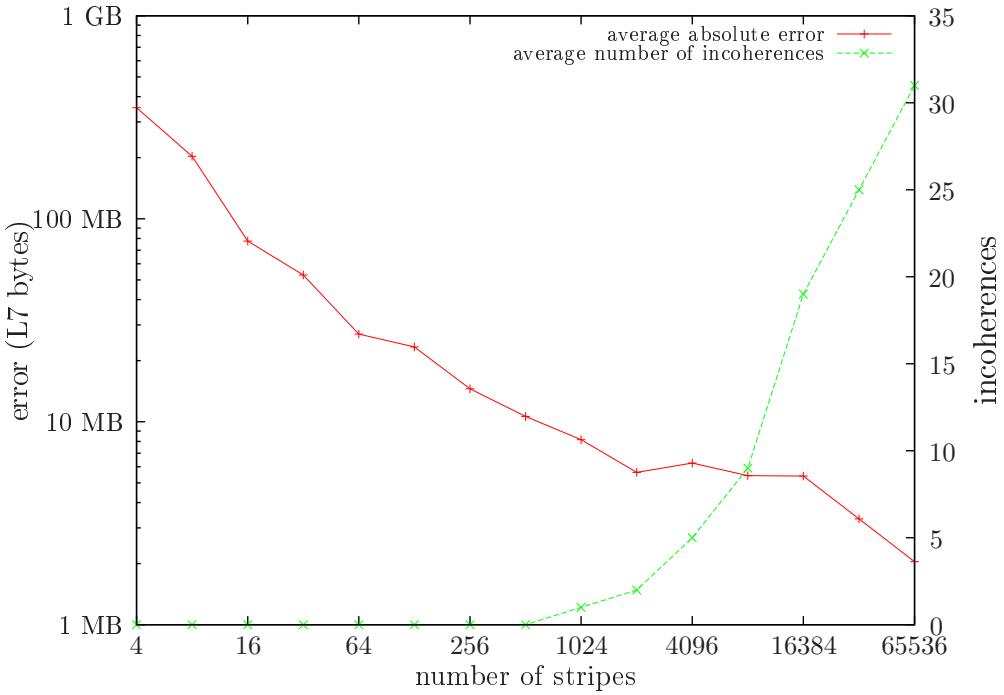


Figure 2.7: Detector Correctness for the Largest Connections

Figure 2.8 shows the performance of the large connection detector. In order to separate the cost of the Secondary Path to that of the Main Path, we ran Bro with the Main Path disabled. For comparison purposes, we also run the Main Path without the Secondary Path, and no application-layer analyzers. The total time required to run the trace was 890 sec.

When the number of stripes is smaller than $S = 1024$, the detector's runtime is basically constant (280 seconds), and only a small part of the time needed by the

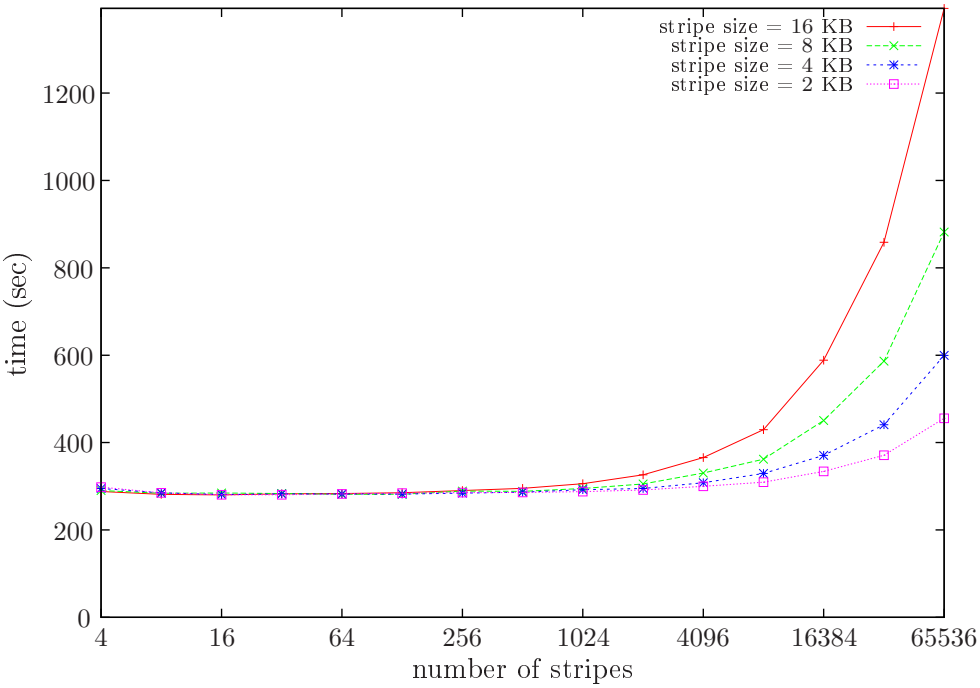


Figure 2.8: Large Connection Detector Performance

Main Path to reject all packets from the trace (890 seconds). Moreover, most of the time is used to to access to the trace (75% of the reported time is system time).

When the number of stripes is larger than $S = 1024$, the running time starts growing significantly, as the amount of traffic processed by the detector grows linearly. (Note that both the x- and y-axis in the figure are logarithmic.)

2.5.3 Heavy Hitters

Rationale

The goal of the heavy hitters (HH) detector is to discover heavy traffic patterns using a low-bandwidth, pseudo-random sampling filter on the Secondary Path.

Related Work

The goal of dynamic traffic characterization is to summarize the traffic in a link by describing the quantitative importance of several categories, while at the same time automatically defining such categories.

Dynamic traffic characterization deals with a slightly different problem than static traffic characterization. In the former, there is an indefinite amount of overlapping categories. A category is interesting if it is large enough in relative terms, and if there are no more specific categories that are just slightly smaller.

In dynamic traffic characterization, the goal is to produce hybrid traffic summaries, in which several overlapping criteria are considered together. The results are filtered so that only the most-specific and significative categories are reported [Estan et al., 2003].

For example, the traffic originated by a host will be interesting, not only if it is large enough, but also if there is not a single connection that accounts for most of its traffic. In the latter case, the interesting item is the traffic from the single connection, which is more specific than the traffic originated by the host.

Autofocus is a tool that automatically characterizes network traffic based on 5 dimensions, namely source and destination address and port, and IP protocol [Estan et al., 2003]. Autofocus starts by computing the most-specific information possible, using the 5 dimensions, which is actually equivalent to per-connection accounting. Then, it selects the categories that exceed some threshold. Last, it compresses

the selected categories by introducing more generic categories, and keeping those significantly larger than the more-specific ones that they comprise.

[Xu et al., 2005] proposes data-mining and information-theoretic techniques to automatically discover significant behavior patterns. Their idea is to start with 4 traffic dimensions (source and destination address and port) and fix one of them (the source address). Then, the traffic is clustered by fixing two of the remaining dimensions, and calculating the uncertainty (entropy) of the fourth. The authors show that the traffic is typically clustered in the extremes (complete uncertainty or no uncertainty at all).

This technique permits creating a behavioral characterization of traffic based on structural models of traffic, simple enough as to permit monitoring of such categories as they change over a period of time.

Heavy-Hitters Detector Description

The goal of HH is to achieve efficient detection of large pseudoflows. We define a *pseudoflow* as a set of packets that share some of the fields in the traditional 5-tuple connection definition (IP source and destination addresses, transport-layer source and destination ports, and transport protocol). Note that this definition includes the connection (packets sharing the 5 fields), but also other cases, as a host being flooded (all packets sharing the same IP destination address field), a busy application server (all packets sharing a common IP address and port value), etc.

The basic idea of the HH detector is very similar to that of Autofocus [Estan et al., 2003]. HH accounts for a traffic stream using the most specific counting criterion, the 5-tuple connection definition, and then tries to compress the information about small connections into more-generic categories. The latter are obtained by the addition of data alongside one or more of the 5 categories. For example, a host scanning a network may not have any large connection, but the addition of all the data corresponding to all its connections may be important enough as to deserve reporting.

HH captures traffic using the Secondary Path, which may be expensive to run in full traces. Therefore, the input of the detector is typically a low-bandwidth, pseudo-random sample of the traffic being monitored.

HH presents several advantages over the large connection detector discussed in Section 2.5.2. First, HH accounts not only for TCP traffic, but also for any UDP and ICMP. Second, HH is able to discover heavy traffic patterns different from connections.

Operation

The Heavy Hitters detector works by obtaining a pseudo-random sample of the traffic monitored, and then clustering all packets that share several network- and transport-layer characteristics together in pseudoflows. These pseudoflows are checked periodically, and when the amount of traffic in any of them is large enough, the detector reports it immediately.

In the normal operation, HH keeps 7 tables with data from connections, as

described in Table 2.2. The *specificity* field is used to order the tables from more specific to more generic. Every time a new packet is received by the detector, the corresponding counter in every one of the tables is updated.

table name	specificity	description
saspdadp	4	connection (traditional 5-tuple definition)
saspda__	3	traffic between a host and a host:port pair
sa__da__	2	traffic between two hosts
sasp____	2	traffic from or to a host:port pair
sa____dp	2	traffic between a host and a remote port
sa______	1	traffic from or to a host
__sp____	1	traffic from or to a port

Table 2.2: Tables Used by the Heavy Hitters Detector

The user may define a series of warning levels. When any of the traffic tables reaches a warning level, an alert is fired. After the alert, all less specific tables are instructed not to consider the data that caused the alert in a future alert.

Note that more specific tables (the most specific being the connection table) have lower warnings levels than more generic ones. Therefore, a very large connection will appear first as an alert in the connection table, and the alerted contents will not be considered for an alert in any of the more generic tables.

We use Bro's state managing capabilities to control the amount of state. All of the 7 tables use self-expiring entries, which are removed when no activity (read or write) has been detected in a fixed amount of time.

Performance

Table 2.3 shows an example of a report from the Heavy Hitters Detector. The first 5 lines are produced in realtime (the time field represents the timestamp when they are produced). The remaining lines are produced after the processing has ended (this only happens in traces). All addresses are anonymized. The *flags* field states whether the reported host belongs to the list of hosts belonging to the internal network being monitored (a user-configurable parameter).

Connection Health Statistics

Alongside the basic pseudoflow activity counters (packets and bytes), we have defined several pseudoflow "connection health" statistics for traffic, including (a) TCP symmetry [Kreibich et al., 2005], (b) ratio of control TCP segments (i.e., segments with their SYN, FIN, or RST flags on) to data (non-control) ones, and (c) ratio of packets to connections. These statistics are used to differentiate legitimate from ill-formed pseudoflows. The first two apply only to TCP traffic, while the latter applies to any type of traffic. Traffic is considered *bogus* if any of the three connection health statistics falls outside of user-defined correctness bounds.

The first statistic used is *symmetry*, defined as the ratio of packets in the forward direction of a connection to packets in the reverse direction of the same connection. The intuition behind this statistic is that a well-formed TCP connection must have traffic in both directions. Even if the application-layer transfer of data is unidirectional,

Table 2.3: Example Report From Heavy Hitters Detector

time	pseudoflow id	pkts	bytes	event	flags
1130965527	164.254.132.227:* <-> *:*	986 k	823 MB	large src	internal
1130969123	*:* <-> 164.254.133.198:80/tcp	1.07 M	654 MB	large dst	internal
1130990210	*:* <-> 164.254.133.194:*	1.12 M	357 MB	large dst	internal
1130992153	54.75.124.72:19150/tcp <-> 164.254.133.146:*	977 k	79 MB	large flow	
1130999627	164.254.132.247:80/tcp <-> *:*	1.02 M	781 MB	large src	internal
	164.254.132.227:* <-> *:*	1.90 M	1.47 GB	large src	internal
	164.254.133.198:80/tcp <-> *:*	1.84 M	1.22 GB	large src	internal
	164.254.132.247:80/tcp <-> *:*	1.21 M	968 MB	large src	internal
	71.213.72.252:80/tcp <-> 164.254.133.56:*	498 k	522 MB	large flow	
	:80/tcp <-> 164.254.132.88:	459 k	479 MB	large dst	internal
	: <-> 164.254.133.194:*	1.35 M	427 MB	large dst	internal

the receiver should be sending back ACK segments to clock the transmitter.

Due to the widespread use of delayed ACKs ([Braden, 1989], Section 4.2.3.2), we expect the symmetry in well-formed TCP connections to be between 0.5 and 2. Considering the effect of packet losses and sampling artifacts, we expect the limits which will cause an alarm event to be more encompassing. [Kreibich et al., 2005] proposes a 8:1 ratio as sensible limit, but they use non-sampled traffic, which produces more accurate symmetry ratios.

Note that this statistic would apply to the 7 tables mentioned, and not only to the connection one. The reason is that all the tables are non-directional, i.e., they account for information about both sides of a connection in the same table entry.

The second statistic is *control/data*, defined as the ratio of TCP control segments (those with the SYN, FIN, or RST flag set) to TCP non-control (data) segments. The intuition behind this statistic is that control segments are signaling traffic, and therefore a large stream of legitimate TCP traffic should be composed of much more data segments than control ones. A very large ratio of control segments compared with data segments is considered a signal of bogus traffic.

The third statistic is *packet/connection*, defined as the ratio of number of packets to the number of connections. Note that this statistic does not apply to the connection table (where the number of connections is always one), but does apply to non-TCP traffic.

The intuition behind the *packet/connection* statistic is that a high-volume pseudoflow

will be composed of a small number of connections when compared to the number of packets. If a pseudoflow has a number of connections comparable to its number of packets, we consider this as a signal of a bad behavior.

Using the tables and the three connection health statistics, we define some heuristics that differentiate pseudoflows in the following categories:

- Flooders and Floodees.

  We define a flood as a *high-volume* host, host:port pair, or host:remote port pair, for which the traffic is bogus. Floods occur typically because of attacks or broken protocols.

  An example of a flood is a "SYN flood," where lots of TCP segments with the SYN bit set are seen by the NIDS. A that uses SYN segments as signals of new connections, and reacts to them by creating per-connection state, can be easily stressed by a SYN flood[4].

  A SYN flood directed to a given host will be seen by the NIDS as a large entry of bogus traffic (the *control/data* ratio will be unusually large) in the host table.

  Flood events can be used to fight floodings behavior, by detecting flooders in real-time, and then using this information to avoid creating state in the NIDS associated to the flooders. A straightforward decision is to blacklist flooders.

[4][Dreger et al., 2004] states that Bro creates up to 240 bytes of state per SYN segment, and that a significant part of these state chunks correspond to SYN segments never answered (e.g. scans). The authors show how a connection compressor permits deferring the instantiation of a full connection state until both sides of the connection have shown activity.

Note that this scheme detects both flooders and "floodees" (hosts being subjected to floods). This helps to limit the event production in the NIDS. For example, if several attackers are flooding a given host, we want to generate a "host being flooded" that encompasses all the packets corresponding to the distributed flood.

- High-Volumers

  We define a high-volumer as a host that is generating or receiving a very large amount of *legitimate* (non-bogus) traffic.

- Big Connections

  We define a big connection as an entry in the connection table with an unusual amount of traffic. An example is a large FTP transfer.

- Big Application Sessions

  We define a big application session as a large entry in "host and a host:port pair" or host:host tables with legitimate traffic. A big application session corresponds to a high-volume transfer (the session) splitted among several connections. This is often used, for example, to avoid the problems of TCP congestion control in very fast links [Lee et al., 2001].

- Non-Symmetric TCP Connections

  We define a non-symmetric TCP connection as a entry in the connection or the host:host tables whose symmetry statistic is too different from 1.

We are still working on tuning the values for the three statistics that differentiate

bogus traffic from valid traffic.

2.5.4 Backdoor Detection

Another application well-suited for implementation using the Secondary Path is Zhang and Paxson's work on how to use packet filters to efficiently detect backdoors [Zhang and Paxson, 2000a]. The authors define backdoors as connections not running in their well-known port, preferentially interactive, and propose several filters to detect them.

[Zhang and Paxson, 2000a] proposes two different mechanisms to detect backdoors: The first one consists of looking for traces of interactive traffic behavior by analyzing the timing characteristics of small (less than 20 bytes of payload) packets. The intuition behind it is that interactive connections will be characterized by short keystrokes (large proportion of small packets) caused by human responses (large proportion of large intervals between each two small packets).

The second one consists of extracting some signatures of particular protocols (SSH, FTP, Gnutella, etc.), and using them to analyze traffic in each protocol's standard port. Finding a server for a given protocol running in a port other than the standard one may indicate the presence of a backdoor.

We have added both approaches to Bro using our Secondary Path model. Doing so is simple, and provides a nice operational capability, namely the ability to detect use of these protocols in a very efficient fashion. It also permits to integrate the

results into further analysis, this time using the Main Path.

Signature-Based Backdoor Detectors

The first addition for backdoor detection are the 8 signature-based backdoor detectors proposed by [Zhang and Paxson, 2000a]. From them, we discarded the rlogin and telnet ones because they are too broad. In the generic trace used for our experiments, 50 K packets match the rlogin signature, and 92 match the telnet one. This coincides with operational experience running the original backdoor detectors in LBL.

In the telnet case, visual inspection of some packets matching the telnet signature shows packets corresponding to bulk data transferences (SSH, HTTP, and other protocols) whose first 2 bytes of payload happen to be 0xff and 0xfa-0xff, respectively.

Implementation

The main advantage of implementing the detectors using the Secondary Path is ease in adding them. Figure 2.9 shows the SSH analyzer code. backdoor_ignore_ports is a set of ports where interactive traffic is expected, including FTP, SMTP, SSH, rlogin, telnet, and others.

The second advantage is that, when coupled with BPF state tables (see Section 3.7) or with Shunting (see Chapter 4), it permits activating the Main Path when a backdoor uses a protocol that the NIDS knows how to analyze. For example, if the

```
global ssh_sig_filter = "
  tcp[(tcp[12]>>2):4] = 0x5353482D and
  (tcp[((tcp[12]>>2)+4):2] = 0x312e or tcp[((tcp[12]>>2)+4):2] = 0x322e)";

event backdoor_ssh_sig(filter: string, pkt: pkt_hdr)
  {
  # get rid of traffic in well-known ports
  if ( ["ssh-sig", pkt$tcp$sport] in backdoor_ignore_ports )
    return;
  if ( ["ssh-sig", pkt$tcp$dport] in backdoor_ignore_ports )
    return;

  print fmt("%s backdoor_ssh_sig,, %s:%s -> %s:%s", network_time(),
      pkt$ip$src, pkt$tcp$sport, pkt$ip$dst, pkt$tcp$dport);
  }

redef secondary_filters += {
  [ssh_sig_filter] = backdoor_ssh_sig,
};
```

Figure 2.9: SSH Backdoor Detector Example

analyzer detects an SSH connection in a non-standard port, it can add a new entry in the BPF table that captures packet from the connection, and label the traffic accordingly so that the Main Path knows it must use its SSH analyzer to process traffic from that connection.

Evaluation

In the evaluation side, we have just focused on the performance, instead of the correctness of the mechanism. The latter is already discussed by [Zhang and Paxson, 2000a].

We ran four different experiments on the *tcp-1* trace:

approach	explanation	time
A	Main Path, no analyzers	890 sec
B	Main Path-based backdoor analyzer	1659 sec
C	Main Path, Sec. Path-based backdoor analyzer	1064 sec
D	Sec. Path-based backdoor analyzer	327 sec

Table 2.4: Performance of Signature-Based Backdoor Detector

A Bro using no application-layer analyzers

B The original implementation of the backdoor code, which is fired by Bro events, and is not filter-based

C The new backdoor code using the Secondary Path

D The new backdoor code using the Secondary Path, after disabling Bro's Main Path

In all cases, we ran all the detectors but the telnet and rlogin one.

Table 2.4 shows the performance of the original and Secondary Path-based detectors.

The extra cost caused by the original, Bro-event based, backdoor detector implementation is 769 sec (B - A).

In comparison, the extra cost of running the same detectors using the Secondary Path is just 174 sec (C - A). The code is basically several pieces of the form depicted in Figure 2.9.

Generic Analyzer

We also wrote the Generic Algorithm for Detecting Interactive Backdoors described by [Zhang and Paxson, 2000a] using the Secondary Path. The code is very similar to the current Bro version of the detector, but being fired by the Secondary Path, instead of Bro events.

The two main advantages of implementing the detector using the Secondary Path are ease and performance. Section A.1 in Appendix A describes the detector implementation.

Correctness

The correctness of the approach is discussed in the original paper [Zhang and Paxson, 2000a]. We compared the results of the original detector and the one based on the Secondary Path. As the used trace (described in Section 2.5.1) had almost no backdoor-like traffic (just some AOL Instant Messenger, or AIM, traffic), we decided to check how good was the detector for discovering the trace's only well-known interactive connections, namely SSH traffic. In order to do so, we took SSH out from the list of well-known ports where the detector does not carry any processing.

Performance

We ran four different approaches on the *tcp-1* trace:

A Bro using no application-layer analyzers

F The original implementation of the generic backdoor code, which is fired by Bro events, and is not filter-based

G The new generic backdoor code using the Secondary Path

H The new generic backdoor code using the Secondary Path, after disabling Bro's Main Path

Table 2.5 shows the performance of the four different approaches.

approach	explanation	time
A	Main Path, no analyzers	890 sec
F	Main Path-based generic backdoor analyzer	1296 sec
G	Main Path, SP-based generic backdoor analyzer	1179 sec
H	SP-based generic backdoor analyzer	284 sec

Table 2.5: Performance of Generic Backdoor Detector

In this case, the extra time incurred by the original detector is 406 seconds, while the extra time incurred by the SP-based version is 289 seconds.

2.6 Conclusions

We have described the Secondary Path, an alternate channel for acquiring packets in intrusion detection and monitoring environments. The Secondary Path supports analyzers interested in isolated packet, network-layer (connection-less) based processing. This is in comparison with analyzers that prefer using the application-layer based processing environment provided by the Main Path.

The Secondary Path rationale is that, in some scenarios, alternate traffic processing based on isolated packet analysis can provide useful information that complements or disambiguates the information obtained from the Main Path.

The Secondary Path permits new filtering models based on packet filtering and sampling. We present several examples, and show how all of them can be easily implemented using a very small amount of code. An added advantage is performance: Secondary-Path detectors run sensibly faster than their Main Path-based counterparts.

Chapter 3

Packet Filter Augmentation

3.1 Abstract

This Chapter describes two new packet filter mechanisms that provide richer, fine-grained, dynamic control of the packet filtering process.

These mechanisms are designed with the goal of keeping the simplicity that defines the popular BPF packet filter, while allowing the filter to both capture traffic at high-speed rates, and perform efficient validation of the filters for security and running-time bounding purposes.

The first mechanism provides in-kernel, packet-based random sampling, introducing randomness as a first-class object in both the human-readable language where users write their filters, and the low-level, assembler syntax-like language that is effectively run in the kernel.

The second mechanism provides in-kernel, fixed-size, generic-purpose, persistent, associative tables plus a set of hash functions to index them. The main goals of this addition are to provide easily-available connection-based random sampling, and to permit implementing the Shunting device (see Chapter 4) in the kernel, without the need of special-purpose hardware.

3.2 Introduction

The traditional definition of a packet filter is a "kernel-resident, protocol-independent packet demultiplexer." [Mogul et al., 1987] A packet filter is a system that receives network traffic, and sends some of the packets to a set of processes that register in the packet filter. Whether each process receives the packets or not, depends on a criterion (the filter) that each process defines. For example, an application may be just interested in all web traffic, while another just wants to see ICMP pings.

When a user-level process wants to receive a subset of the traffic arriving to a host, one option is to receive all the traffic, and then decide which packets are interesting. This mechanism is very flexible, but it forces each packet into crossing the kernel-user boundary, even if it is not needed by the application. If the actual number of packets of interest for the user-level process is low, the approach results in a large system overhead.

A key idea in packet-filter systems is the *filter*. A filter is an application-specific program that, when run against a packet, returns a Boolean value that states whether

the packet is interesting to the application or not (this is also known as whether the filter accepts or rejects the packet). Instead of the applications receiving all packets and then picking the ones they are interested, applications can provide the kernel with a filter that lets the kernel take the same decision, therefore only dispatching interesting packets.

The fact that filters are run on the application's behalf by the kernel permits that packets that are not wanted by any application never cross the kernel-user boundary. In some scenarios, this results in a large performance savings.

Running user-provided filters in the kernel presents two tradeoffs, namely security and unbounded running times. Malicious or broken programs may obtain access to privileged resources, cause mayhem in the hosts where they are running, or just run forever, effectively hogging the host resources. The kernel must therefore be able to validate the filter safety, i.e., that the program will not do anything that it is not supposed to do, and to bound the total amount of time a filter may run.

The traditional approach to address both issues consists of a) writing the filters in a special-purpose, low-level language limited to performing packet filtering; this language is normally simple enough so as to permit fast filter running, cheap validation of the filter safety, and efficient calculation of a bound in the running time; and b) running the filter a virtual processor with very limited resource access.

Running packet filters in high-speed environments stresses the performance issues in the packet filter. Any added mechanisms must follow the general principle of

minimizing the amount of per-packet work carried out. This normally translates into simplicity. At the same time, this efficiency concern cannot be dealt with without consider the impact on security.

In this Chapter we discuss the justification, implementation, and performance of new packet filtering models, based on two abstractions that augment current packet filters. These abstractions are randomness and persistent state, and they are designed with the efficiency and security concerns in mind.

The proposed additions intend to provide the following new filtering models:

- sampling: a primary need in network monitoring is sampling, both packet- and connection-based. Sampling permits sound analysis of traffic streams without the requirement of parsing them in full. This is useful in scenarios where full analysis is impossible or too expensive. For example, a client may be interested in studying a set of network packets, so large that its full analysis is out of reach. If the properties being studied remain constant for a sample of the set, then analyzing the set may provide similar answers. Randomness is therefore a crucial need.

- persistent state: the second addition provides the capability to keep state between different packets in the packet filter. We provide persistent-state management facilities that are a) secure, as they limit the amount of state available to each user process, its access to privileged resources, and guarantee easily-provable bounds on the running time; b) simple enough to permit fast

implementation; and c) generic enough to support the addition of other new filtering models of which we are not aware.

Our persistent-state management mechanisms also permits the use of filters that modify their persistent state themselves[1], without the need for their applications to request the changes explicitly. This feature permits managing filter state without the need to cross the kernel-userlevel boundary.

- richer filter control: the last augmentation consists of having richer control of the packet filtering process. We provide an efficient mechanism that enables dynamic, fine-grained control of the filter program. This is translated in two filtering models.

  First, support of efficient management of incremental filters where stateful parsing of application-layer traffic is required to refine the filter specification. For example, if you want to capture all FTP traffic in a link, capturing all packets that correspond to the standard FTP port just captures control connections [Postel and Reynolds, 1985]. Data connections often use negotiated, random ports different from the standard FTP control port. Therefore, they will not be capture by a filter that focus on the latter. On the other hand, it is possible to monitor the negotiation in the control connections, therefore obtaining information that uniquely identifies the data connections. It is easy for an application to

[1] This does not imply self-modifying filters, which is a hard problem from a security point of view, and whose usefulness is not clear. Filters may modify their state, but not their program.

parse the control connections, and when it detects the negotiation of a data connection, to add it to the filter. We support this by providing efficient access to the filter persistent state.

Other examples of protocols where the bulk-data connections are dynamically negotiated in control connections include multimedia session control protocols [van der Merwe et al., 2000] and p2p protocols [Karagiannis et al., 2004].

The second filtering model includes connection-based sampling. Connection-based sampling has different properties than packet-based sampling [Duffield et al., 2002], and may be useful depending on the use the samples will have.

An outline of this Chapter is as follows: Section 3.3 discusses related work. Section 3.4 introduces a simple random packet-sampling mechanism, and discusses its implementation. Section 3.5 compares our pseudo-random sampling approach with a simple, cheap solution used to imitate it, so that the strengths and weaknesses of the latter can be understood. Section 3.6 shows some results from pseudo-random sampling. Section 3.7 introduces inter-packet state in packet filters. Finally, Section 3.8 summarizes the chapter.

3.3 Related Work

This Section classifies related work in three different categories, namely packet filters (Section 3.3.1), packet classifiers (Section 3.3.2), and sampling (Section 3.3.3).

3.3.1 Packet Filters

A *Packet Filter* is a mechanism to select packets from a packet stream using a programmable criterion (the filter).

Operation

Traditional packet-filter implementations use a simple processor (often virtual) operated by a reduced, special-purpose Instruction Set Architecture (*ISA*). This simple processor, which typically resides inside the kernel, has full access to the packet being filtered, which is mapped into the processor memory.

Packet filters operate as follows: Applications register a filter with the packet filter. When the packet filter receives a new packet, it runs every registered filter against the packet. Every filter that accepts the packet causes the packet to be dispatched to the corresponding application. This operation mode enables several client applications with registered filters at the same time.

Applications express their filter needs using a high-level language, which is compiled into a lower-level language (the ISA language) that can be run in the packet-filter processor. The rationale for the two-tier language system is threefold. First, applications can write filters using a human-readable language consisting of packet-filtering specific primitives combined with Boolean operators. This high-level language syntax permits flexible expression of complicated filters. Second, the packet-filter processor receives filters written in a low-level language which is simple enough to be run fast, enables

validation of the filter safety (which is important for both malicious and broken filters), and permits bounding the filter's running time. Third, this two-tier system permits code optimization in the compilation process between the high-level and the low-level languages.

CSPF

CSPF [Mogul, 1990; Mogul et al., 1987] proposed the idea of putting a pseudo-machine language interpreter in the kernel, which avoids filtered-out packets crossing protection boundaries. CSPF parses a high-level filter description into a Boolean expression tree, which then is run through each packet using an operand stack-based interpreter. Both constant values and bytes obtained from the packet are pushed into the stack. A set of arithmetic and logical operations pop the top two words from the stack, and then push back the result. After evaluating a program, if the top of the stack is not zero, or the stack is empty, the packet is accepted. Otherwise it is rejected.

CSPF presents several shortcomings. First, an operand stack is not the usual CPU model. Therefore, it must be simulated, which is quite inefficient as it requires a lot of memory accesses. Second, the encapsulated nature of network protocols makes the tree model inherently inefficient, as it cannot express protocol dependencies. And third, CSPF cannot parse variable packet headers, as there is no indirection operation.

BPF

In order to address these shortcomings, BPF proposes substituting the Boolean expression tree with a directed acyclic Control Flow Graph (CFG), and the stack-based interpreter with a register-based virtual machine [McCanne and Jacobson, 1993].

BPF is the most common packet-filter architecture today, and is used as the development framework for most packet-filter research, including this Chapter. Therefore, we describe it in depth.

BPF filters are written using a high-level language (called expressions) composed of packet-related primitives linked by Boolean operators. Primitives usually consist of an identifier (name or number) preceded by one or more qualifiers. There are three different kinds of qualifiers, namely type, dir, and proto. Type qualifiers select what the identifier refers to. There are three types of type qualifiers, namely "host", "net", and "port". Dir qualifiers specify the transfer direction to/from the identifier. There are two types, "src" and "dst". Proto qualifiers specify a particular protocol. There are several proto qualifiers, including "tcp", "udp", "ip", and others. Valid Boolean operators are "and", "or", "not".

As an example, the expression "src port 80" matches all packets whose source port is 80. The expression "ip[2:2] = 60" matches all IP packets where the result of concatenating the second and third IP bytes (the IP length field) produces the value 60.

High-level expressions are actually compiled into low-level language programs,

which are then run in the BPF virtual machine. This machine consists of a buffer with the packet contents, two registers (A and X), a small scratch memory (denoted M[]), and an implicit program counter.

Low-level language programs are based on a reduced instruction set. The BPF ISA follows an "assembler syntax" composed of six different instruction types:

1. load instructions: copy a value into either A or X. Addressing modes (where the value comes from) include immediate, direct (fixed offset) from M[] or the packet buffer, indirect from M[] using X, and the packet length. Load instructions include ld (load word, or 32 bits), ldh (load halfword, or 16 bits), ldb (load byte, or 8 bits), which load the corresponding packet data into A; and ldx (load word), which loads the corresponding packet word into X.

2. store instructions: copy either A or X into M[]. Load instructions include st (store word), which stores the word in A into M[]; and stx (store word), which stores the word in X into M[]. Both store instructions operate using indirect addressing with a constant value.

3. ALU instructions: this category includes both arithmetic and logic instructions that get its operands from A and either X or a constant, and put the result back into A. ALU instructions include add, sub, mul, div, and, or, lsh, and rsh.

4. branch instructions: alter the program counter depending on the result of a comparison test between A and either X or a constant. Note that branch

offsets are always positive values, so only forward branches are allowed. Branch instructions include jmp, jeq, jgt, jge, and jset (the latter performs a conditional bit test).

5. return instructions: terminate the filter and indicate how many bytes of the packet to dispatch to the applications. (Length zero means the packet is rejected.) The only return instruction is ret.

6. other instructions: the original implementation includes just transferences between A and X in this category. This includes tax and txa.

Inter-Packet and Inter-Filter State

An interesting detail in BPF is all runs of a packet over a filter are completely independent of each other. Every packet filter uses its own scratch memory (denoted M[]), so two different filters do not affect each other. We call this property "inter-filter independence."

Also, BPF has no persistent state. The scratch memory is zeroed every time there is a new packet, so the processing of a packet in a filter does not affect the processing of further packets in the same filter. We call this property "inter-packet independence."

BPF is inter-filter and inter-packet independent.

MPF

MPF [Yuhara et al., 1994] describes two modifications to BPF to make it more efficient when used for carrying out network- and transport-layer protocol processing in microkernels.

The first modification introduces sharing of some processing in the same packet among several filters. For efficiency reasons, protocol processing in microkernels is carried out outside the kernel. This means that each connection requires registering a filter in the packet filter. Given that each packet must be run with every registered filter, plain protocol processing causes a large overhead in the microkernel.

As all protocol stacks use very similar filters, of the form "get all packets that correspond to the {TCP/UDP, remote address, remote port, local address, local port} tuple", MPF adds to BPF an associative match function that permits combining together all stack filters. This way, all stack filters are run using a single filter, that eventually dispatches to a single corresponding application (the stack that must receive the traffic).

The second modification affects the way IP fragments are processed. One of the main problems of packet filters is how to process IP fragments. IP fragments do not include transport-layer headers, and therefore its source and destination ports are unknown. When a fragment arrives to a microkernel, it must be dispatched to all connections whose remote address is the packet's source address, independent of the source port. It is the responsibility of the user-level protocol stack to detect whether

the fragment corresponds to its remote port or a different one before accepting or dropping it.

MPF proposes the addition of per-filter, persistent memory to link fragment information (the IP ID field) to the transport-layer information (the source and destination ports). It was the first to suggest the use of inter-packet dependencies.

xPF

xPF [Ioannidis et al., 2002] proposes taking BPF's idea of pushing packet processing to the kernel to a deeper level. xPF's goal is for the packet filter to be not only a mechanism for demultiplexing applications to user-space, but also an engine to fully execute monitoring applications.

While keeping the BPF model, xPF proposes two main additions to it: a) persistent memory: The authors propose to add inter-packet persistent memory, which can be read and written by the compiled filter and the user-space application. It also proposes adding indexed load and store instructions, which requires careful checking for every instruction and every running of the engine. b) eliminating engine restrictions: In order to provide a richer execution environment, xPF proposes eliminating BPF's requirement of forward-only branches. This introduces a potential starvation problem (a user-specified filter getting into an infinite loop, for either malicious or bad-programming reasons), which the authors tackle by limiting the number of BPF instructions any filter can run. If a filter goes beyond its assigned instruction budget, xPF runs

a user-specified exception handler, which retains the original forward-only branch requirement.

mmdump

mmdump is a tcpdump-like tool that is able to efficiently capture multimedia-session connections [van der Merwe et al., 2000]. The main problem in capturing multimedia streaming flows (RTSP, H.323) using plain BPF is that the ports used to send the bulk data are not fixed. Instead, they are dynamically negotiated during the session control connection, which does use a fixed, well-known port.

A multimedia-session connection capturer must therefore listen to session control connections, parse their contents, and dynamically modify the kernel filter to add any new, negotiated data connection.

In order to capture the data sessions, mmdump adds two new features to BPF. The first one is dynamic modification of filters. Whenever information from a new data session is distilled from a session control connection, it must be added to the compiled BPF filter that already resides in the kernel. The straightforward approach of changing the tcpdump filter every time there is an addition, then recompiling and installing it, is very expensive, inefficient, and does not scale with the number of tracked sessions. Instead, the authors take advantage of the similarity and simplicity of all data session descriptions ("host A and port B"), generating a new BPF subtree per session description, and grafting it to the overall kernel filter.

The second new feature is persistent state. While it is not necessarily required in order to capture the data sessions, mmdump uses per-session state to a) keep session statistics (which are reported on session teardown), and b) buffer the full session control data before sending it to the corresponding protocol parser.

Lastly, mmdump uses a resource scarcity detector to force the eviction of persistent, per-session state.

FPL

FPL [Cristea and Bos, 2004; Cristea et al., 2005] is a new packet-filter language intended to run, among others, in the FFPF network monitoring framework [Bos et al., 2004; Nguyen et al., 2004]. While FFPF can run several other packet-filter languages (including BPF), FPL is designed to exploit all of FFPF features.

FPL is based on registers and persistent memory. It uses a new, generic (including for loops, if/then/else operations, and a native hash function), fully functional, extensible language as the high-level language. It uses the processor native language as the low-level one. The latter means that it can be run natively in the kernel, but also in special-purpose hardware, as network processors or dedicated ASIC boards. This should provide an important performance boost, as compared to generic boards. On the other hand, implementing a generic language in dedicated hardware introduces performance concerns related to efficient pipelined implementations.

FPL deals with security issues at both language levels. When compiling a high-level

language program, FPL limits the number of times a for loop can be run. This is done in order to ensure bounds on the number of instructions run by a filter. When they finish compiling a filter, trusted FPL compilers add an MD5 signature to the filter. This should ensure that the FPL processor only runs low-level (native) language programs produced using a trusted compiler.

Efficient Packet Filter Compilation

A related part of work on packet filters concerns the optimization of the filtering process.

DPF [Engler and Kaashoek, 1996], among other code optimizations, proposes the generation of dynamic code from the filter description, so that the filter can be compiled and run instead of interpreted.

BPF+ [Begel et al., 1999] enhances the optimization of the filtering by detecting and eliminating redundant predicates and computation, and by looking for opportunities to use lookup tables when carrying out several comparisons related to the same field. BPF+ also permits compiling the high-level filter specification into native code, by using just-in-time compilation, and checks the security of the natively compiled filter by using a code verifier.

3.3.2 Packet Classifiers

Closely related to packet filters are packet classifiers. A packet classifier is a mechanism that inspects packets from a packet stream, and determines how to process them. The processing action can be where to dispatch a packet, which resources (e.g. priority) to assign to a packet in an OS, or how to route a packet in a router.

Packet classifiers work by assigning a tag to each packet, and an action to each different tag. Compared to packet filters, packet classifiers associate packets with richer semantics than just "filter in" or "filter out". On the other hand, their decisions are typically limited to parsing a group of selected fields.

Packet classifiers are typically specified using declarative actions, known as "rules". Packet classifiers are normally programmed by declaring a set of rules, which map patterns to tags. When a packet is received, it is matched against all the patterns, and the matching one is used to select the tag. By comparison, packet filters are specified using imperative code.

The non-exclusive nature of packet classifier rules poses the problem of conflict detection and resolution, in other words, what to do with a packet that matches several patterns. Packet classifiers solve it by selecting always the longest prefix that matches.

Another difference between packet filters and packet classifiers is that the latter are composed of a very large set of rules, in some cases more than several tens of thousands rules. Packet filters, on the other hand, tend to work with more concise

rules.

In order to support classifying traffic in high-speed networks, PathFinder [Bailey et al., 1994] proposes a design to express pattern-matching specifications. PathFinder supports fragmentation by storing information on how to process a packet from the first fragment, and out-of-order fragments by postponing their processing. PathFinder describes two implementations, one in software and one in hardware. It proposes the implementation of a subset of the classifier rules in hardware. The network adapter runs a subset of the rules against the packet. If the packet matches any, it is (quickly) classified and processed without ever reaching the host. Otherwise, it is sent to the host for complete processing.

3.3.3 Sampling

Another large piece of related work is packet and connection sampling. Claffy *et al.* [Claffy et al., 1993] discuss three packet sampling methods, namely systematic sampling, stratified random sampling, and simple random sampling, and study their effects in the distribution of packet sizes and packet interarrival times.

In (simple random) packet sampling, the sampler makes a random decision on whether to sample each packet or not. Packets are classified in connections, which can also be sampled. In connection sampling, the sampler only takes a random decision for the first packet of the connection. After that, all packets corresponding to the same connection follow the same decision.

Note that, assuming the same sampling ratio, a packet has the same probability of being sampled whether using packet or connection sampling.

Packet sampling is easier to implement than connection sampling, as the latter requires the sampler remembering the decision on whether packets from a given connection must be sampled or not. On the other hand, it is not possible, in general, to obtain connection statistics from packet sampled traces, as packet sampling removes some of the flow information present in the unsampled stream.

Duffield *et al.* [Duffield et al., 2002, 2003] discuss how to determine connection statistics from packet-sampled traces, and which connection properties from the original stream can be inferred from packet-sampled connection statistics.

Estan and Varghese [Estan and Varghese, 2002] propose the use of "sample and hold" and multistage filters to efficiently estimate statistics of large flows. Estan *et al.* [Estan et al., 2003] show a method to automatically cluster traffic: Their idea is to start with exhaustive (except in the case of IP addresses) information about the traffic in the leaves of a tree, and keep compressing the tree upwards until they reach a given threshold.

3.4 Random Sampling

The first modification is the introduction of randomness, and as a consequence, random sampling. We consider pseudo-random sampling a major necessity in kernel packet filters. Pseudo-random sampling is not available in BPF.

Random packet-sampling is a relatively straight-forward addition to packet filters. We provide a new instruction that provides clients a Pseudo-Random Number Generation (*PRNG*), that can be instructed to provide a number in a user-provided range. The PRNG also permits the user to seed it directly. This can also be used to enforce deterministic behavior, which is useful for debugging purposes.

3.4.1 Implementation

A new Pseudo-Random Number Generator (PRNG) has been written as an extension to BPF. It produces pseudo-random, uniformly-distributed values in the range 0 to $2^{32} - 1$.

The tcpdump expression syntax (see Section 3.3.1) has been extended with a new type qualifier, "random", that generates a load instruction with random addressing mode. For example, the result of running the primitive "random(x)" is a random number between 0 and $x - 1$. The expression "random(3) = 0" returns the Boolean true value randomly, with probability 1 in 3.

The BPF ISA (see Section 3.3.1) has been extended with a new load command, namely ldr. ldr is implemented as a new addressing mode for the BPF load instruction. When a load instruction with the addressing mode set to random is run, the BPF engine loads into the selected register a random value between 0 and the value of the instruction immediate value (k field).

Randomness Source

Randomness is produced using two different PRNGs, including a) a popular Linear Congruential Generator [Park and Miller, 1988], and b) a less-popular, but stronger random number generator based on the RC4 algorithm [Schneier, 1995].

The main advantage of the LCG generator is that it is widely available. We wanted to know whether performance was an issue. We ran a simple experiment, in which packets from a large trace were sampled with a very low sampling rate, using a simple filter "random(x) = 0". The rationale of this setup can be justified as follows: First, we used a large trace and a very simple filter that consumed one random number from the PRNG per packet. This maximizes the number of times the random generator is called. Second, the tcpdump expression produced a very low sampling rate. This minimizes the influence of the inherent per-packet processing in the BPF engine.

The performance differences between processing the full trace with the LCG-based generator versus the RC4-based generator were negligible (around 0.1%).

In any case, it is not clear how an attacker could break the LCG generator: it cannot access its internal state (neither the seed nor whether a given packet is sampled or not), and the packet-filter capturing process does not produce any outgoing activity that can be used to sense its state [Kumar et al., 2005].

Random Seeding

Another feature of the PRNG is that the user is able to specify the seed, so they can have added control over the randomness (say, to thwart attackers from guessing the state). We have implemented this in BPF by adding two new methods in the standard device/socket control mechanism (ioctl /setsockopt).

Table 3.1 shows the codes for access to the random seed in the ioctl case.

command	explanation
BIOCSRNDSEED	set the random seed
BIOCGRNDSEED	get the current random seed

Table 3.1: ioctl API to Configuring PRNG

We considered adding random seeding as a new expression primitive that is run only once. The idea was to prepend a primitive like "random_seed(0xdeadbeef)" in front of the primitive using the randomness (e.g., "ip and random(10) = 0"). When the kernel processed the expression for the first time, it would notice the run-once "random_seed" primitive, run it (effectively seeding the PRNG), and then remove it from the per-packet filter program.

The main advantage of this approach is easy portability. Random seeding is requested using the already-existent control mechanism (i.e., the expression). Its implementation only requires modifications on the expression parser (so that the new primitive is recognized as a valid one), and the BPF engine (so that the seeding is actually carried out, and then removed from the filter). Both modifications are common to all packet filter implementations. In particular, it does not require adding

a new control channel to the kernel (the ioctl /setsockopt calls described above).

The idea of the initialization, run-only-once primitives could also be extended to create a model that supports other generic one-time initializations, such as a private key for the hashing operands (see Section 3.7).

We found dynamic filter adjusting to be too cumbersome. First, BPF is too rigid for this. For example, BPF does not recognize numbers bigger than 4 bytes. This is a problem to some of the initializations we have in mind, as for example when initializing an *HMAC* key, which is 16-bytes long.

Second, we want to be able to reuse an already-compiled filter with a different seed.

Finally, the reduced BPF instruction space seems the wrong choice to support potentially indefinite configuration options.

Optimizer Issues

The main implementation difficulties are related to the BPF optimizer. First, BPF considers itself free to arbitrarily reorder primitives, which may change the expression semantics.

For example, this means an expression like "port telnet or (port ftp and random(3) = 0)", which means "get any telnet traffic, or any ftp traffic with 1/3 probability", might get reordered to "get any telnet traffic, or with probability 1/3 look to see if it's FTP, and if so capture it", which has very different sampling properties.

Second, BPF is keen to collapse two equal instructions with no dependencies. For example, if the tcpdump filter reads twice the same IP field, the optimizer will use the value of the first reading in both uses. This is wrong for PRNG, as two calls to "random(x)" with the same value of x should produce independent result. The solution is to make any 2 random instructions different to the optimizer eyes. We do this by marking each instruction with the address of the memory used to store the instruction itself.

3.4.2 Random Sampling Behavior

This Section contains results of a very simple experiment to check the correctness of the random sampling addition (RND) in a controlled environment. Just to make sure the addition works, we run the filter "random(x)", for different values of x, on a packet trace, and counted the number of packets captured as a function of the packet being processed.

Figure 3.1 shows the number of packets captured by RND as a function to the packet being processed. We report results for three different sampling rates, namely 10, 1000, and 4096 to 1. Both the x and y-axis are in a logarithmic scale.

We can see in all the cases how RND captures the expected ratio of traffic rapidly.

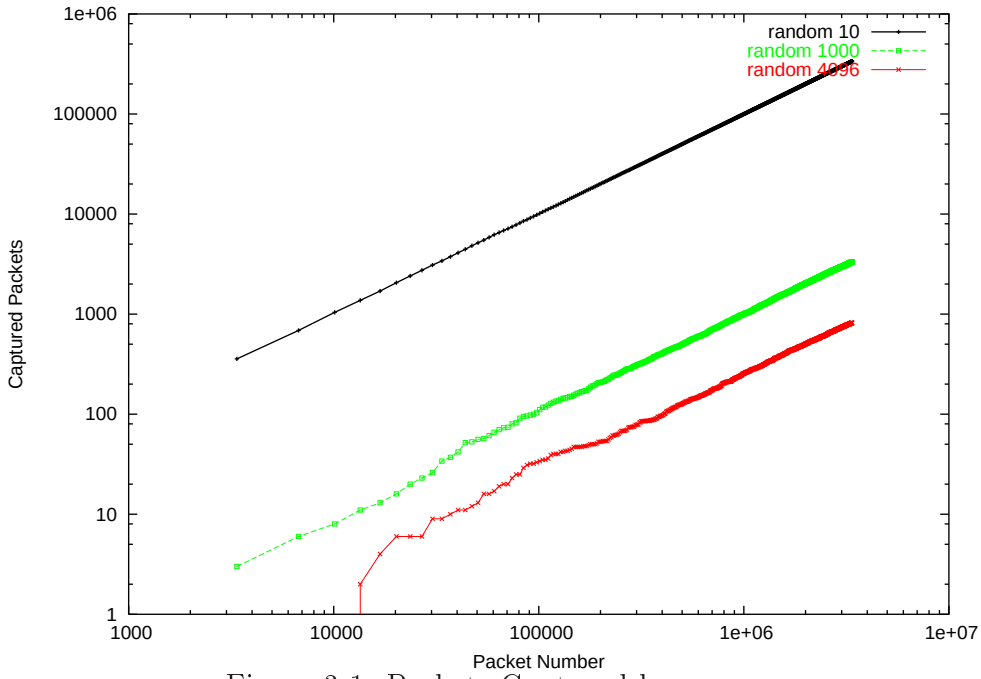


Figure 3.1: Packets Captured by RND

3.5 Random Sampling Discussion

In this Section, we compare the pseudo-random sampling approach, RND, with a simple, currently-available approach, SAMP.

The SAMP approach tries to leverage on the inherent entropy of IP headers to carry out unbiased sampling. SAMP assumes the IP checksum field to be approximately a random number. Sampling is carried out by masking the IP checksum field, and comparing the result to a fixed number. As an example, if you want to sample 1 in 4 packets, you could use the tcpdump expression "ip[10:2] & 3 == 0". This expression gets the last 2 bits of the 11th IP header bit (i.e., the last 2 bits of the IP checksum less significant byte), and returns true if their value is zero.

The main benefit from SAMP is portability. It can be used in any BPF-based packet filter. Users need not modify the Operating System kernel to sample a packet stream.

Note that cost is not really an issue. Masking and comparing two numbers is not that different from generating a random number using only integer operations (for example, an LCG-based PRNG requires an integer multiplication and division per random number). In a simple experiment, the differences in the time taken to run SAMP versus RND on the same trace, and with the same capture ratio, where less than 1%.

This Section compares both approaches, assuming RND produces a sample with statistical properties comparable to that of a true random sample. We want to know in which scenarios is SAMP OK, and in which ones SAMP is wrong.

The remainder of this Section is structured as follows: Section 3.5.1 discusses the entropy present in IP headers. Section 3.5.2 lists the approaches that different Operating Systems take in order to manage the value of the IP ID field, and Section 3.5.3 studies the consequences of each of these approaches in the "randomness" of the IP checksum field values, and therefore the quirks shown by SAMP when capturing traffic from each of the approaches. Section 3.6.1 provides evidence of such quirks by running both RND and SAMP in some isolated, unsampled traces. Section 3.6.2 describes a comparison of the performance of RND to SAMP in a real environment for a large amount of time. Section 3.6.3 concludes.

3.5.1 On IP Header Entropy

Figure 3.2 depicts the fields in an IP header.

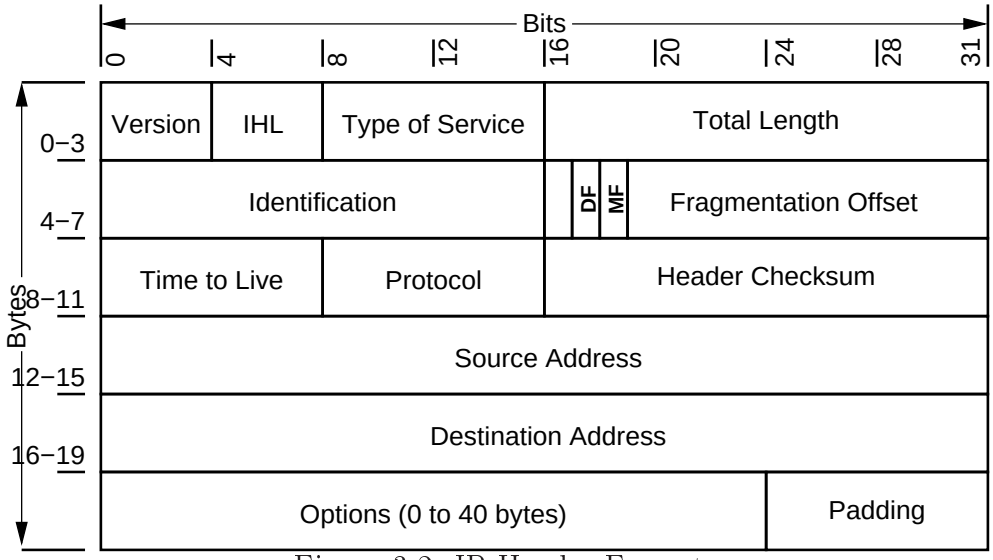


Figure 3.2: IP Header Format

The IP checksum field (bytes in positions 10 and 11) is produced as a linear combination of all other IP header fields (actually, as the negation of the 1's complement sum of all the other fields [Postel, 1981a]). Assuming there is enough entropy in any of the remaining IP header fields, SAMP should produce relatively good samplings.

How good is such an assumption? Let's consider the main packet aggregation unit, i.e., the connection. All packets from a connection share the same source and destination addresses, protocol, and typically TTL fields. Moreover, IP options are very rare, which means the header length field almost always has the same value. The TOS/Diffserv/ECN field currently goes unused most of the time, or with the same value for every source host.

The packet length is a better source of entropy, but not good enough. A large connection is likely to be used to transfer bulk data (e.g., a large file) or live multimedia contents (e.g., a VoIP conversation). In the first case, the packet size is likely to be always the maximum transmission unit (MTU) in the data transfer direction, and the minimum transport protocol size in the opposite (acknowledgment) direction. A less-common use of large connections is sending multimedia content. In this case, we must differentiate Variable Bit Rate (VBR) encoders from Constant Bit Rate (CBR) ones. The latter will most likely use constant-sized packets.

The IP packet length is a better source of entropy for small connections, with one exception: TCP control segments usually are the same size for all packets between the same two hosts. The reason is that TCP control packets do not carry TCP payload, and the number of TCP options is typically the same in all packets between the same two hosts.

Fragmentation could be another source, but in our traces is very uncommon: less than 0.2% of our packets are fragments. Other researchers report similar numbers [Shannon et al., 2002].

Therefore, the principal source of entropy when calculating the IP checksum field is the IP ID field.

3.5.2 IP ID Field

The IP Identification (*IP ID*) field is a 16-bit long field used to implement IP fragmentation [Postel, 1981a]. It is used to "distinguish the fragments of one datagram from those of another." [Postel, 1981a] The only requirement for setting the IP ID field is to "set (it) to a value that must be unique for that source-destination pair and protocol for the time the datagram will be active in the internet system." [Postel, 1981a]

As fragmentation is uncommon, the field is typically unneeded. Its value, though, varies depending on the IP stack implementation. We have seen at least the following behaviors in a significant number of hosts. (Note that the approaches do not necessarily exclude each other.)

- CONSECUTIVE: One per-host counter, increased by one in each packet. This is the most common approach, as it is the one used by Windows hosts.

  A variation of this approach in some little-endian processors does the increment in network order, so the actual increase is 256 (as is the case in some old Windows hosts). This does not affect the rest of the discussion, so we will consider both cases together under the same label.

  Another variation includes one per-flow counter, increased by one in each new packet. A "flow" is defined as all the packets between the same endhost pair. Note that the variation includes per-flow counters, instead of per-connection

counters. The reason is that the IP ID field can only depend on other IP fields. Ports are transport-layer fields, and fragments (except the first one) do not carry transport headers. This variation is preferred by several Unix flavors (e.g. Linux 2.4.21.).

- ZERO: In some cases, hosts set the IP ID field to zero for some packet subset, and using other approaches for the remaining packets (e.g. *CONSECUTIVE*).

  The most common variation of this approach consists of using the *CONSECUTIVE* approach for all packets but the TCP SYN/ACK segments. These segments are always sent with the IP ID field set to zero.

  The goal of setting SYN/ACK segments to zero is to avoid leaking information on the number of packets a host sends, which can represent a privacy problem [Bellovin, 2002]. Some flavors of Linux 2.4 and 2.6 follow this behavior.

  In order to set the IP ID to zero, the source end-host needs to ensure that packets will not be fragmented in their path to the destination. Note that setting the DF ("Don't Fragment") bit in the IP header is not enough: If a middlebox receives a DF packet so large that it cannot be forwarded, the middlebox will discard the packet [Postel, 1981a].

  Methods to ensure no fragmentation will occur fall into two categories: First, some hosts set the IP ID field to zero only in packets that are small enough to avoid fragmentation. This variation is followed by some older Linux (2.4) and

FreeBSD versions (5.2-RC2). Second, other hosts use Path MTU to discover the maximum transmission unit (MTU) along the path [Mogul and Deering, 1990], and then set the IP ID field to zero in packets whose size is smaller than the MTU.

Sending SYN/ACK segments with zero IP ID fields is an example of the first category: these segments are always very small packets (typically 40 bytes plus the TCP options), and it is very unlikely that they will need fragmentation.

Other hosts actually set the IP ID fields to zero in TCP segments with the RST or FIN flags set, as they do not carry payload, and therefore are limited to the IP and TCP header size.

Note that setting the IP ID to zero is a bad idea. Some middleboxes (routers, load balancers, firewalls, etc.) illegally remove the DF along the path (e.g. Cisco DSL/L2TP recommends avoiding this in its trouble shooting section for broadband access via DSL/L2TP, where the MTU is lower than 1500 because of the tunneling overhead).

- RANDOM: Some OpenBSD and FreeBSD flavors have the option of making the IP ID field random for security reasons. FreeBSD assumes that getting a random counter per new connection is an expensive task compared to the importance of the information leakage, and therefore it is disabled by default.

3.5.3 Influence of the IP ID Field Behavior in SAMP Features

The existence of different approaches to IP ID management will produce three main effects in the traffic sampled by the SAMP approach.

Effect 1: Because of the prevalence of *CONSECUTIVE* stacks, most large connections are sampled systematically, instead of randomly.

Let's consider a large connection using *CONSECUTIVE*. As we mentioned already, almost all packets from that connection will have the same IP header field values.

The only field that therefore changes between packets in the same large connection, and therefore its only source of entropy when calculating the IP checksum, is the IP ID. This field is increased just by one on each consecutive packet.

SAMP only captures a packet when the packet's checksum matches a given mask. In other words, when its value is in a fixed set of values, which we will call the "checksum set". As the function used to calculate the IP checksum is actually linear [Postel, 1981a], the "checksum set" will have a corresponding "IP ID set" of values that will cause a packet to be sampled by SAMP.

If the IP ID always changes by +1, this means that SAMP will not sample packets pseudo-randomly. Instead, SAMP will sample packets systematically, picking one packet of every $R = 1/p$, where p is the sampling ratio.

Systematic sampling has different properties than random sampling [Lohr, 1999].

Let's consider a flow i, composed of n_i packets, and captured using a p sampling

ratio. If the flow is captured with the RND approach, the capture of each packet is a Bernoulli process, and therefore the total number of packets captured follows a Binomial distribution with parameters n_i, p. The average and variance of the number of captured packets are shown in Equations 3.1 and 3.2.

$$\mu_{i,random} = n_i p \tag{3.1}$$

$$\sigma^2_{i,random} = n_i p(1-p) \tag{3.2}$$

What is the aggregated effect of random sampling? Let's assume the stream is composed of F flows, named f_1, f_2, $\cdots$, f_F, each composed of n_i packets, and where $N = \sum^F n_i$. The sampling of flow i is independent of the sampling of flow j, where $j \neq i$. Therefore, the average and variance of the aggregated number of packets sampled from the full stream are show in Equations 3.3 and 3.4.

$$\mu_{T,random} = \sum^F \mu_{i,random} = \sum^F n_i p = pN \tag{3.3}$$

$$\sigma^2_{T,random} = \sum^F \sigma^2_{i,random} = \sum^F n_i p(1-p) = Np(1-p) \tag{3.4}$$

Now let's try to get the same statistics for SAMP, which does systematic sampling. Let's assume the number of packets in the ith flow is $n_i = k_i R + \epsilon_i$, where k_i and ϵ_i are integers, and $0 \leq \epsilon_i < R$. A systematic sampler will capture k_i packets with

probability ϵ_i/R, and $k_i + 1$ packets with probability $(R - \epsilon_i)/R$. The average and variance of the number of captured packets for flow i are shown in Equations 3.5 and 3.6.

$$\mu_{i,systematic} = \sum^{F} x_i p_i = (k_i + 1)\frac{\epsilon_i}{R} + k_i\frac{(R - \epsilon_i)}{R} = k_i + \epsilon_i p = n_i p = \mu_{i,random} \quad (3.5)$$

$$\sigma^2_{i,systematic} = \sum^{F}(x_i - \mu^2_{i,systematic})p_i = (k_i + 1 - n_i p)^2\frac{\epsilon_i}{R} + (k_i - n_i p)^2\frac{(R - \epsilon_i)}{R} =$$

$$= (1 - \epsilon_i p)\epsilon_i p \quad (3.6)$$

Note that the variance $\sigma^2_{i,systematic}$ is bounded between 0 and 1/4. As $0 \leq \epsilon_i/R = \epsilon_i p < 1$, the variance $\sigma^2_{i,systematic} = (1 - \epsilon_i p)\epsilon_i p$ will be $0 \leq \sigma^2_{i,systematic} \leq 1/4$.

What is the aggregated effect of systematic sampling? Let's consider the same stream as before. The sampling of each flow in the stream is independent of each other. Therefore, the aggregate number of packets sampled from the full stream is $\mu_{T,systematic} = \sum^F \mu_{i,systematic} = \sum^F n_i p = pN$ on average. The variance will be $\sigma^2_{T,systematic} = \sum^F \sigma^2_{i,systematic} = \sum^F (1-\epsilon_i p)\epsilon_i p$, which is bounded by $0 \leq \sigma^2_{T,systematic} \leq F/4$, where F is the number of flows.

A per-flow comparison shows that the average number of packets captured by both SAMP and RND is the same ($\mu_{i,systematic} = \mu_{i,random}$). This also applies to the aggregated results ($\mu_{T,systematic} = \mu_{T,random}$).

The per-flow variance is smaller in the SAMP case than in the RND case. In effect,

applying $n_i = k_i R + \epsilon_i$, $0 \leq \epsilon_i < n_i$, and being ϵ_i an integer, $\epsilon_i = 0$ or $\epsilon_i \geq 1$, and therefore $\epsilon_i p \geq p$. If $\epsilon_i = 0$, then $\sigma^2_{i,systematic} = 0 < \sigma^2_{i,random}$. If $\epsilon_i \geq 1$, then $\epsilon_i p \geq p$ implies $1 - \epsilon_i p \leq 1 - p$, and therefore $\sigma^2_{i,systematic} < \sigma^2_{i,random}$ (see Equation 3.7.)

$$\frac{\sigma^2_{i,systematic}}{\sigma^2_{i,random}} = \frac{(1-\epsilon_i p)\epsilon_i p}{n_i p(1-p)} = \frac{(1-\epsilon_i p)}{1-p}\frac{\epsilon_i}{n_i} < 1 \tag{3.7}$$

As the per-flow variance is smaller, $\sigma^2_{T,systematic} = \sum^F \sigma^2_{i,systematic} < \sum^F \sigma^2_{i,random} = \sigma^2_{T,random}$, and therefore the aggregated variance is also smaller in the SAMP case than in the RND case

In summary, the aggregate results of SAMP will be equal to the results of RND on average, but with a smaller variance (actually bounded in the SAMP case).

Another consequence of sampling being systematic instead of random is that packet inter-timings are biased. Therefore, scaling down inter-packet timings from sampled traces to estimate unsampled inter-packet timings should be avoided.

Considering that most hosts today run some Windows flavor, we should expect the *CONSECUTIVE* approach to manage IP IDs to be almost ubiquitous.

Effect 2: Because of the existence of *ZERO* stacks, some subsets of traffic are aliased.

The *ZERO* approach consists of setting the IP ID field always to zero. If a host follows the *ZERO* approach, the only source of entropy for the packets it sends to a

given destination (a "flow") is the IP length.

The IP length is not a good source of entropy. As we mentioned in Section 3.5.1, large connections are likely to have a constant packet size. Therefore, packets from a flow originating at a *ZERO* host will always have the same IP headers, including the IP checksum field.

If the flow's IP checksum value happens to be one that matches SAMP's mask, all packets from the flow will be captured. Otherwise, no packet from the flow will ever be captured.

Aliasing is commonly defined as a sampling effect that causes two different signals to become indistinguishable when sampled. In our case, the signals are the total number of packets per connection, and the sampled signal is the number of per-connection packets captured by the sampling method.

Let's assume a sampling ratio of 1 in R packets, and two connections. The first one is composed of n packets, and its IP ID field follows the *CONSECUTIVE* approach. As we have seen before, the average number of captured packets will be $n/R = m$.

The second connection is R times smaller, i.e., composed of $m = n/R$ packets. Its IP ID field follows the *ZERO* approach, and its (constant) IP checksum field is part of the "checksum set" of values that are captured by the sampling mechanism. Therefore, all of its m packets will be captured.

The sampling mechanism captures approximately the same number of packets for both connections (m), and therefore the estimation for the original size is approximately

the same in both cases, namely $mR = n$. For the first connection the estimation is fine, but in the second case is completely wrong. We say that the second connection is aliased. More concretely, we say that the connection is aliased *positively*.

If the second connection had a (constant) IP checksum field not part of the "checksum set," no packets from this connection would have been captured. The sampling method would estimate the size of the connection as zero. We say the connection is aliased *negatively*.

In relative terms, the negative versus positive aliasing ratio would be a function of the sampling ratio, p. Let's assume SAMP is sampling 1 in R packets, where $R = 1/p$. If a flow is aliased, whether it is positively aliased or negatively aliased depends on the mask chosen for SAMP. The mask is matched proportionally to p. If we consider R *ZERO* flows, we will expect 1 to be aliased positively, and the remaining $(R - 1)$ to be aliased negatively. This means that, from all aliased flows, the ratio of negatively-aliased flows will be $q = (R - 1)/R = 1 - p$, and the ratio of positively-aliased flows will be $1 - q = p$.

What is the effect of each type of aliasing on SAMP? When SAMP is used to capture a negatively-aliased, n packet-long flow, it will capture no packets from it. This means n packets will not be seen by SAMP. When SAMP is used to capture a positively-aliased, n packet-long flow, it will capture the whole n packets from it. After scaling back the sampling ratio, SAMP would report that it captured n packets from a $n/p = nR$ packet-long flow.

What is the aggregated effect of aliasing? A flow is aliased or not independent of whether other flows are aliased.

If a flow is aliased, it will be negatively-aliased with probability q (and therefore not captured at all), and positively-aliased with probability $1 - q$ (captured in full). Therefore, the average number of packets captured will be as shown in Equations 3.8 and 3.9.

$$\mu_{aliasing} = \sum x_i p_i = 0q + n(1-q) = np \tag{3.8}$$

$$\begin{aligned} \sigma^2_{aliasing} &= \sum (x_i - \mu^2_{aliasing}) p_i = (0 - np)^2 q + (n - np)^2 p = \\ &= n^2 p(1-p) = n\sigma^2_{random} \end{aligned} \tag{3.9}$$

The aggregated effect of negative and positive aliasing in SAMP is actually zero on average terms. On the other hand, the variance of the number of captured packets by SAMP is n times bigger than the same variance when packets are captured by RND, where n is the size of the unsampled flow. Therefore, the standard deviation of the number of packets in aliased flows captured by SAMP will be $\sqrt{n}$ times bigger than when captured by RND.

The larger variance means that the total number of packets from *ZERO* stacks that SAMP captures will be "noisier" than the total number captured by RND.

Operating Systems that use *ZERO* are uncommon, so the overall effect should

be small in absolute terms. Moreover, we expect that, in general, R will be a large number, so that the number of positively-aliased flows will be a small fraction of the number of negatively-aliased ones. Negative aliasing will be uncommon, and will slightly decrease the amount of traffic captured by SAMP. Positive aliasing will be extremely uncommon, but its existence may bias the sampling results very significantly, increasing the total amount of captured traffic in a significant way.

A possible solution to avoid aliasing is to further refine SAMP, excluding any packets with zero IP ID. We call this approach NIDZ. By filtering out packets with zero IP ID, NIDZ should get rid of positively aliased flows, therefore avoiding biasing the total amount of captured traffic in a significant way.

As a separate fact, we note that aliasing will be, in general, an asymmetric effect. Aliasing depends just on the sender end-host, which is the one that sets the IP header fields. Therefore, flows can be aliased in one direction, the reverse, or in both.

Effect 3: Because of the selective use of *ZERO* stacks, sampling of some subsets of the traffic is biased.

Let's consider a source host that uses *CONSECUTIVE* to set the IP ID field for all the packets but for TCP SYN/ACK segments, where it uses *ZERO*.

When sending a SYN/ACK segment to a given host, all the IP header fields will be always the same, and therefore they will have the same IP checksum. This causes an aliasing effect, in which, depending on the exact value used in the SAMP comparison,

either all the SYN/ACK segments to a given host, or none, are captured. In other words, the SAMP approach is completely biased against some IP pairs, and completely biased toward other IP pairs.

As we will see later, the prevalence of this approach, at least in the traces we are using, is important enough that, when capturing SYN/ACK segments, the results of the SAMP or NIDZ approaches are too biased to permit drawing valid conclusions from TCP SYN/ACK segment analysis.

3.6 Random Sampling Experiments

3.6.1 Isolated Trace Analysis

This Section shows the three effects in a controlled scenario. The main data set for this study is a trace taken at the International Computer Science Institute on November 2004. The trace is around 45 minutes long, and adds up to 757 connections, 1.9 M packets, and 968 MB (509 bytes/packet on average). The trace is composed of SSH traffic only.

We sampled the trace using both RND and SAMP. In both cases, the sampling ratio was 1 in 4096 packets.

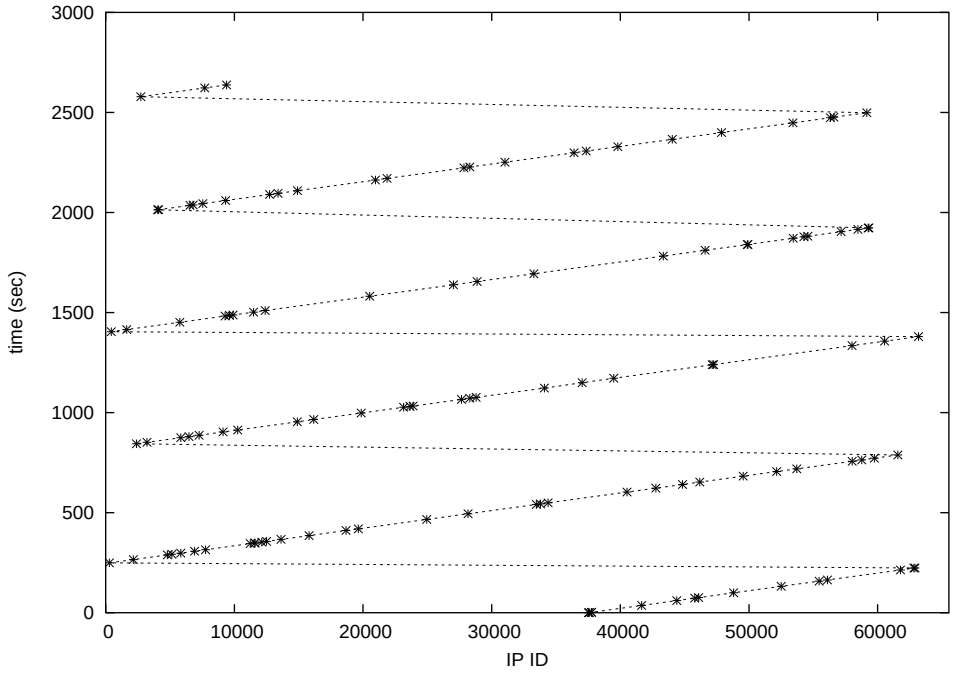
Effect 1: Because of the prevalence of *CONSECUTIVE* stacks, most large connections are sampled systematically, instead of randomly.

Figure 3.3 shows the life of the biggest connection in the trace. The top figure shows the RND-captured connection, while the bottom figure shows the same connection, but captured by SAMP.

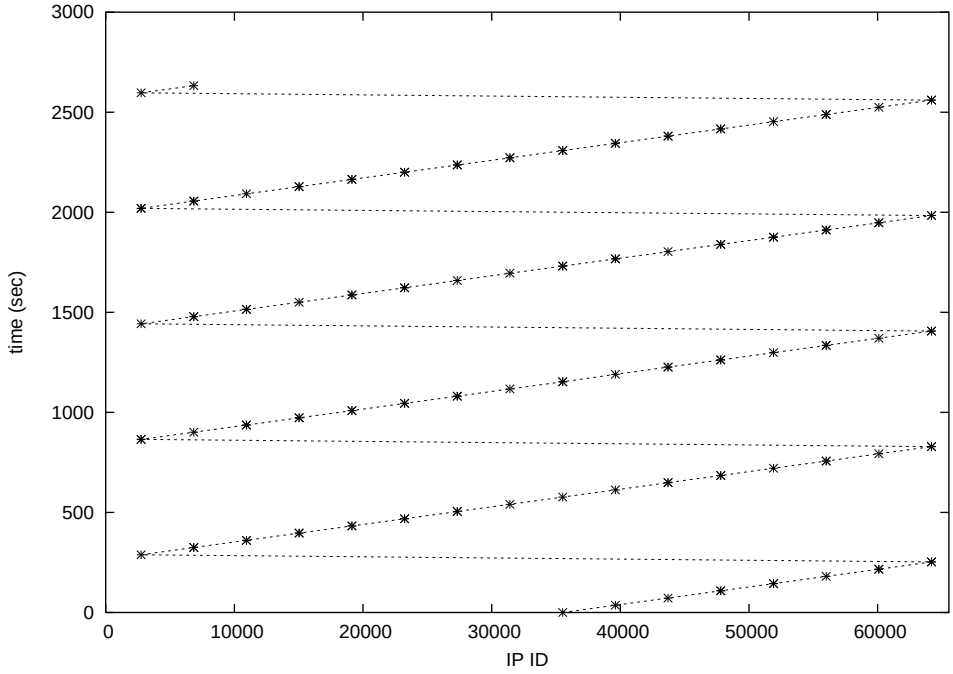
Every point in the trace represents a packet, with the x axis showing its IP ID, and the y axis representing its timestamp. The line stitching all the points together is strictly increasing in the y axis. In other words, it is a time line. We can see the strong systematic 1-in-N effect in the SAMP-captured connection. Almost all packets in both graphs are 1500-byte long.

Figure 3.4 shows the same systematic sampling effect as Figure 3.3, but for the reverse path. Large data transmissions tend to be asymmetric, as one of the hosts sends data to the other, which just keeps acknowledging it. Pure ACKs normally are the same size, accounting for just the IP and TCP headers. While the use of TCP options is common, all packets from the same connection that come from the same host typically have the same number of TCP options. This means all the reverse-path packets have the same size (e.g., 40 bytes when no TCP options are used), and therefore the same systematic 1-in-N sampling effect can be seen in the reverse path. Almost all packets in both graphs are 52-byte long (TCP options account for 12 bytes per packet).

Using both SAMP and RND, we sampled the largest connection, which is composed

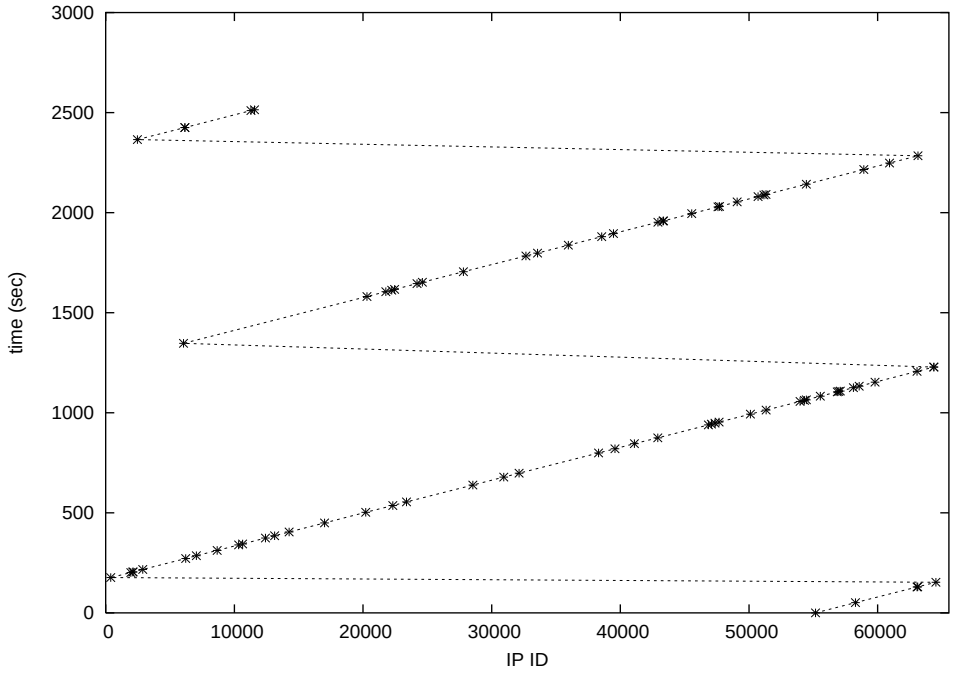


(a) As captured by RND

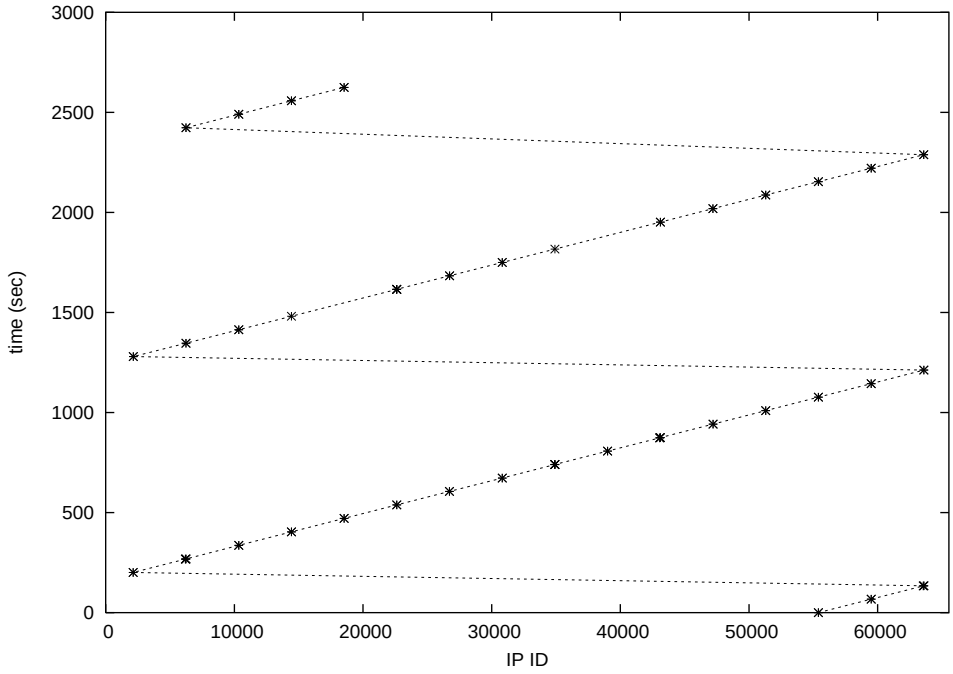


(b) As captured by SAMP

Figure 3.3: Timeline for Biggest Connection in Analyzed Trace



(a) As captured by RND



(b) As captured by SAMP

Figure 3.4: Timeline for Biggest Connection in Analyzed Trace (Reverse Path)

of 465203 packets and 441 MB (24% and 45.6% of the respective totals). In both cases the sampling ratio was $p = 1/4096$, and the experiment was repeated 100 times. Results are shown in Table 3.2, where $\bar{Y}$ represents the sample mean, S^2 the sample variance, and $SE(\bar{Y})$ the standard error of the sample mean.

approach	$\bar{Y}$	S^2	$SE(\bar{Y})$
SAMP	113.51	6.111	0.247
RND	113.44	121.18	1.101

Table 3.2: Packets Captured from the Largest Flow

Note that, while the sample means are very similar, the sample variance of the systematic approach is 19 times smaller than the sample variance of the random approach.

Using both SAMP and RND, we sampled the full trace. In both cases the sampling ratio was $p = 1/4096$, and the experiment was repeated 100 times. Results are shown in Table 3.3.

approach	$\bar{Y}$	S^2	$SE(\bar{Y})$
SAMP	469.13	250.17	1.582
RND	472.44	506.75	2.251

Table 3.3: Packets Captured from the Full Trace

Note that the sample means are again very similar. On the other hand, the sample variance of the systematic approach is only 2 times smaller than the sample variance of the random one.

Note also that, comparing the packets captured from the largest flow and the packets captured for the full trace, the sample variance for the RND approach gets

multiplied by 4. This makes sense, as the ratio between the total number of packets in the full trace and the total number of packets in the largest flow is also 4. On the other hand, the same sample variance for the SAMP case gets multiplied by 41. Flow aggregation in the SAMP case increases the "noise" of the capturing process.

Effect 2: Because of the existence of *ZERO* stacks, some subsets of traffic are aliased.

Our trace does not show any negatively- nor positively-aliased flows. We will look for evidence of such flows in the larger traces experiments.

Effect 3: Because of the selective use of *ZERO* stacks, sampling of some subsets of the traffic is biased.

In our SSH trace, there are 779 TCP packets with the SYN and ACK flags set both to one. From those packets, 669 (86%) have IP ID set to zero, and therefore they have no source of entropy. Their checksum is only a function of the source and destination host. Therefore, the results of sampling TCP SYN/ACK segments using SAMP will be biased toward some IP address pairs, and biased against all the others.

3.6.2 Long-Term Traces

Description

Data sets for this study were collected from a GigEthernet link at Lawrence Berkeley National Laboratory (LBNL). For both the RND and the SAMP sampling approaches, we used an off-the-shelf FreeBSD host. For SAMP, we captured the traffic running plain tcpdump/libpcap/BPF. For RND, we captured the traffic using our modified version of the same capture suite. In both cases, the sampling ratio was 1 in 4096 packets.

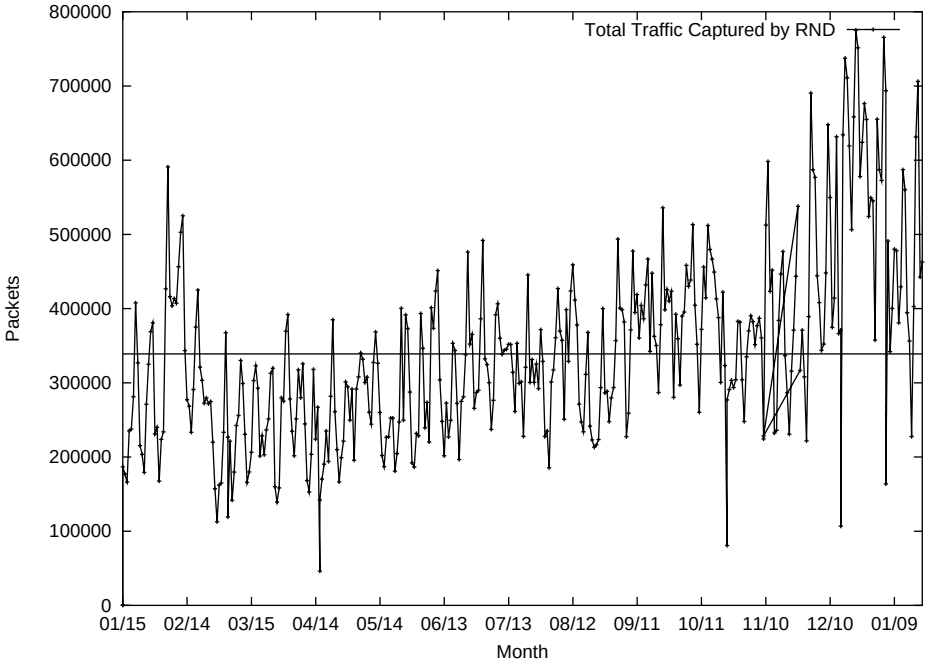
The data sets cover the LBNL DMZ for almost the full year of 2004 and the first two weeks of 2005. Just about all traces are 86,400 seconds (1 day) long, though a few are shorter due to reboots.

Figure 3.5 shows the total amount of traffic captured by RND [2]. The top figure shows the daily results in packets, and the bottom one shows the daily results in bytes. The daily average is 340 K packets (300 MB) per day, which after considering the 1:4096 sampling, produces an average number of 16 K packets/sec (114 Mbps).

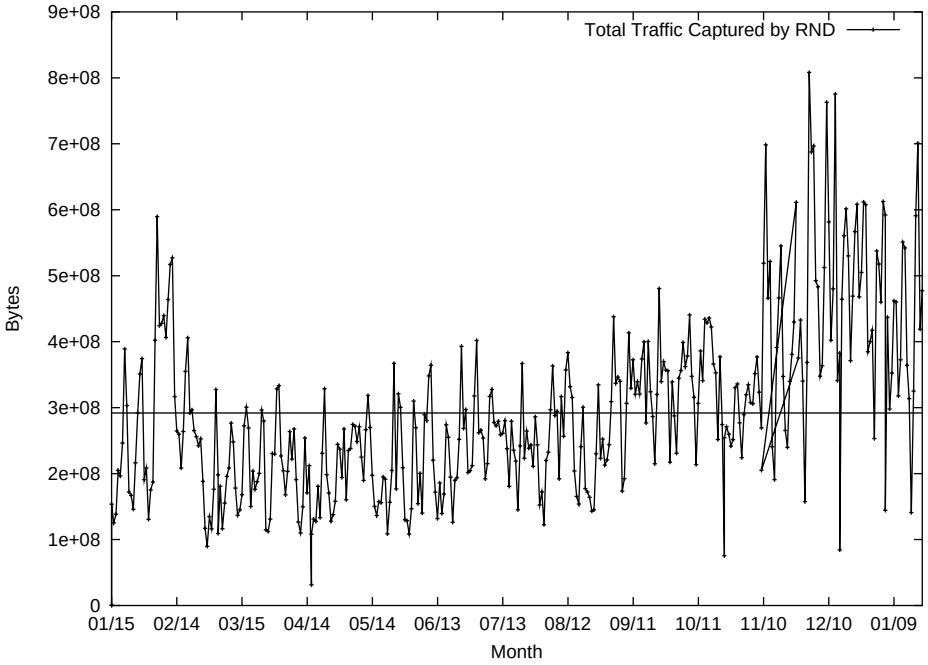
Total Differences

Figure 3.6 shows a) the difference between the amount of traffic captured by SAMP and the amount of traffic captured by RND, and b) the difference between the amount of traffic captured by NIDZ and the amount of traffic captured by RND. The y axis

[2]Note that for the trace description we focus on RND, instead of SAMP, because our approach considers RND to be the correct measurement unit against which SAMP's correctness will be compared.



(a) Packets



(b) Bytes

Figure 3.5: Total Traffic Captured by RND

shows this difference as a percentage of the RND traffic. The top figure measures the traffic in packets, while the middle and bottom ones do the same in bytes and bytes/packet, respectively. A 5% value in the top graph, for example, means that for the given day, either SAMP or NIDZ captured 5% more packets than RND.

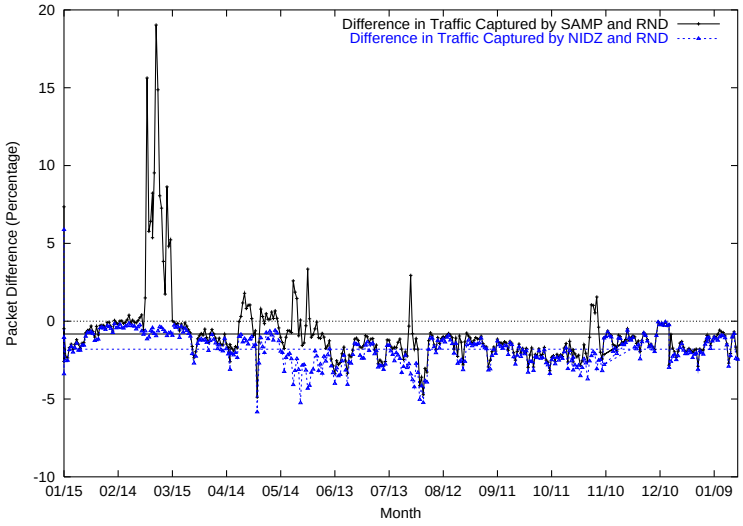
Just considering the total traffic captured, SAMP performs quite well. On average, it captures around 1% fewer packets than RND, and approximately the same number of bytes. The difference in the number of packets is between ±5% at any time except during the first two weeks of March, when the difference increases up to 20% (03/06 trace). The difference in the number of bytes is always between ±10%.

While the amount of traffic is very similar, the traffic being captured is not.

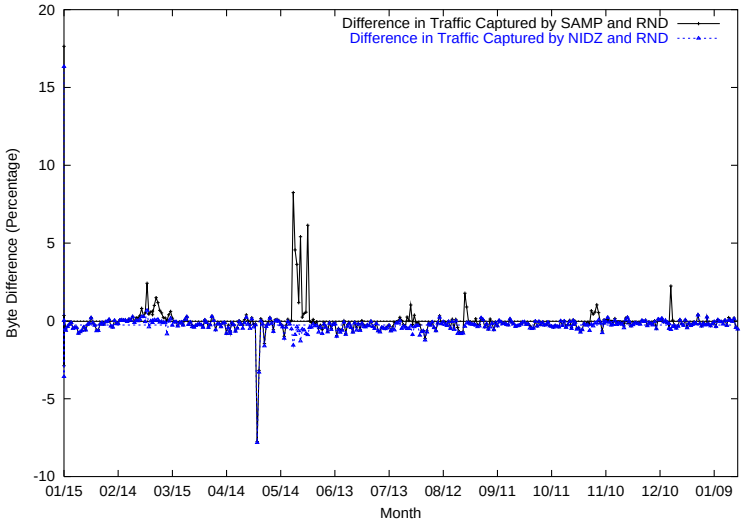
Partial Differences with SAMP

The most obvious difference between RND and SAMP is the 20% extra packets captured by SAMP during the first two weeks of March. Even more interesting, the difference in bytes during the same period is almost imperceptible, and therefore the average packet size is up to 15% smaller (03/06 trace). SAMP is not only capturing more packets, but it's capturing different packets. This is evidence that SAMP is biased toward some types of traffic.

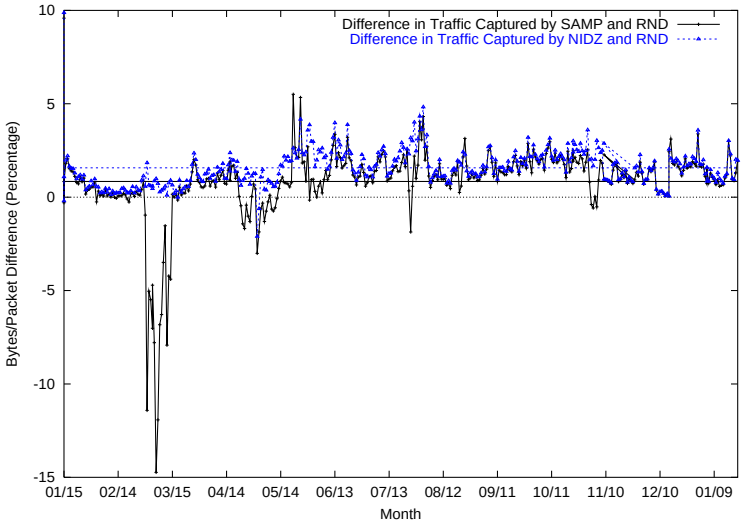
What is causing up to a 20% increase in packets during the first two weeks of March? Based on a visual inspection, we noticed that the excess traffic captured by SAMP, as compared with RND (up to 28,000 packets/day) can be completely attributed



(a) Packets



(b) Bytes



(c) Bytes/Packet

Figure 3.6: Difference in Traffic Captured Between SAMP and RND, and SAMP and NIDZ

to a group of TCP SYN/ACK segments sharing exactly the same IP headers, including a zero IP ID. In comparison, the trace of the same day as captured by RND only shows five packets with the same characteristics.

This is a case of positive aliasing (Effect 2). The increase in traffic, as seen by SAMP, during the first two weeks of March, can be explained by positive, asymmetric aliasing of an IP-pair flow where the source host follows the *ZERO* approach for some subset of the traffic. Because all SYN/ACK segments between this IP pair share the same IP headers, they all have the same IP checksum field, and therefore SAMP captures all of them, instead of a 1:4096 sample.

Correcting Aliasing With NIDZ

Section 3.5.3 introduced NIDZ, a refined version of SAMP in which packets with zero IP ID are filtered out.

The main advantages of NIDZ are (1) it is as simple as SAMP (it just requires adding "and not ip[4:2] = 0" to the filter used to capture traffic), and (2) it avoids positive aliasing by getting rid of packets with zero IP ID. The main disadvantage is that it keeps biasing against some hosts and some subsets of traffic.

Figure 3.6 shows the effect of capturing traffic using NIDZ. Considering the number of captured packets, NIDZ captures on average 2% fewer packets than RND. Its results are, nevertheless, less noisy than those of SAMP. The number of packets captured by NIDZ never differs from the number captured by RND by more than 5% (compare

with 20% in the case of SAMP and RND).

Considering captured bytes, NIDZ follows very closely the results of RND. The main exception is an 8% decrease in total bytes captured in the 05/01 trace.

By analyzing that trace, we can see that 78% of the packet difference and 95% of the byte difference can be explained by UDP packets between a given host and 23 others, all with the same IP headers (1500 bytes length, zero IP ID, and DF bit set).

This is a case of negative aliasing (Effect 2). The decrease in traffic, as seen by SAMP, in the 05/01 trace, can be explained by negative, symmetric aliasing of traffic between a given host and 23 others, where the 24 hosts follow the *ZERO* approach for some subset of the traffic.

Note also that NIDZ does not combat negative aliasing, only the positive one.

Bias Measurement on Traffic Subsets

The long-term traces also present evidence of bias in the sampling of some subsets of the traffic (Effect 3). In order to show this bias, we have defined some subsets of the traffic, and then measured the percentage of packets that have zero IP ID on them.

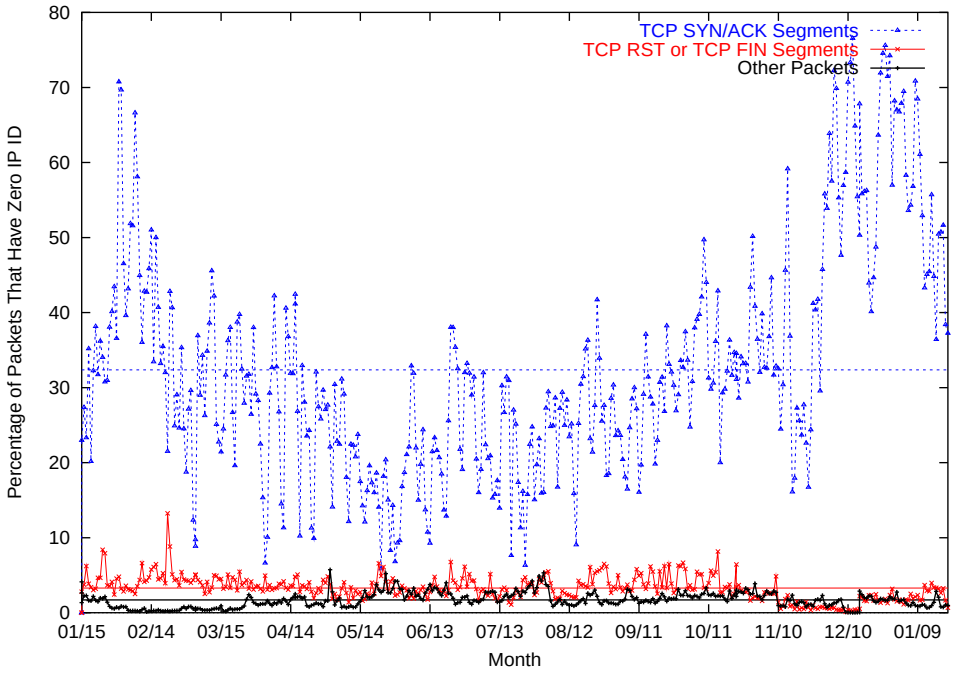
Figure 3.7 shows the percentage of packets with zero IP ID for three different subsets of traffic, namely (1) TCP SYN/ACK segments, (2) TCP RST or TCP FIN segments, and (3) Other packets (any packet not included in (1) or (2)). The top figure shows the results for traffic captured by RND, and the bottom figure shows the

results for traffic captured by SAMP.

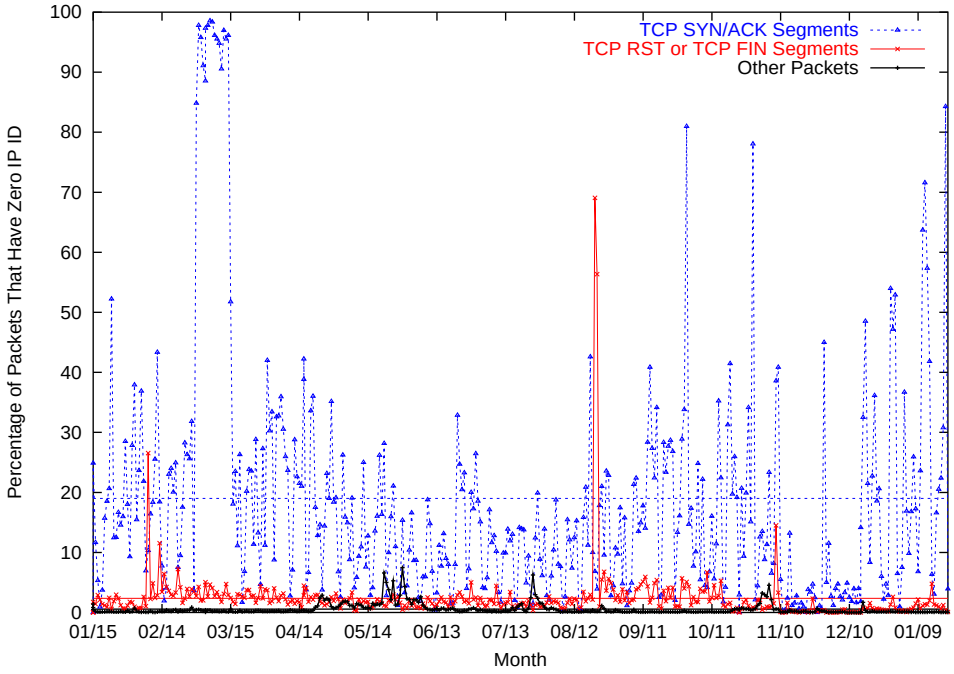
Note the strong discrepancies between the results returned by SAMP and RND, especially in subset (1). This indicates that, for this subset of traffic, the results returned by SAMP are completely biased, and do not represent a sample of the total population. Note that, in this case, using NIDZ does not enhance the sampling quality, as no zero IP ID packets are captured. NIDZ is therefore completely biased against subset (1) of the traffic.

Moreover, from the top figure, we can see that, on average, 33% of TCP SYN/ACK segments have zero IP ID. Assuming that, for pure *CONSECUTIVE* hosts, the probability that a TCP SYN/ACK segment has zero IP ID is negligible (1/65536), then this means that, in 33% of the TCP connections, the responder (the server in a client-server connection) uses the *ZERO* approach to set packet IP IDs. This approach can be either full (i.e., all packets have zero IP ID), or just limited to the TCP SYN/ACK segments. As the percentage of "Other Packets" (subset (3)) that have zero IP ID is negligible, we know that this 33% of the servers use *ZERO* just for the TCP SYN/ACK segments.

As would be expected from the low popularity of hosts running Operating Systems that use it, the pure *ZERO* approach is marginal. Its influence in the aggregated sampling numbers is minimal. On the other hand, an startling third of all the servers use a variation of *ZERO* in which the IP ID of TCP SYN/ACK segments is set to zero.



(a) When Captured by RND



(b) When Captured by SAMP

Figure 3.7: Packets With Zero IP ID

Another subset of the traffic where there is a high percentage of packets with zero IP ID is UDP. Of all UDP packets, on average 23% have zero IP ID. This means that the results obtained from sampling UDP traffic with SAMP are strongly biased. When using NIDZ, we will not see 23% of the total UDP traffic.

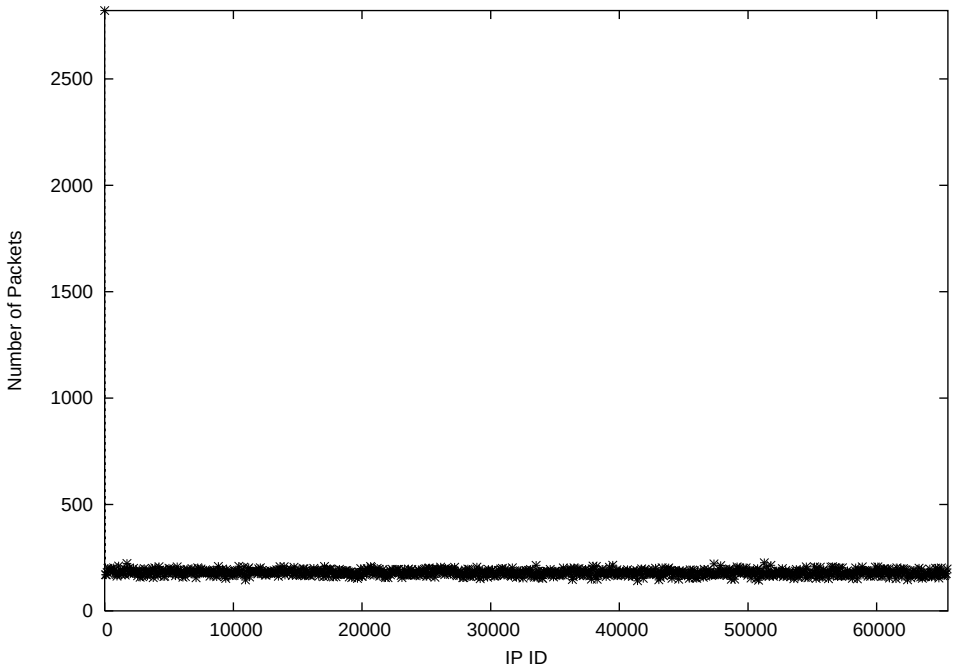
Note that UDP is the main source of IP ID traffic: 78% of all packets with zero IP ID are UDP.

Systematic Sampling

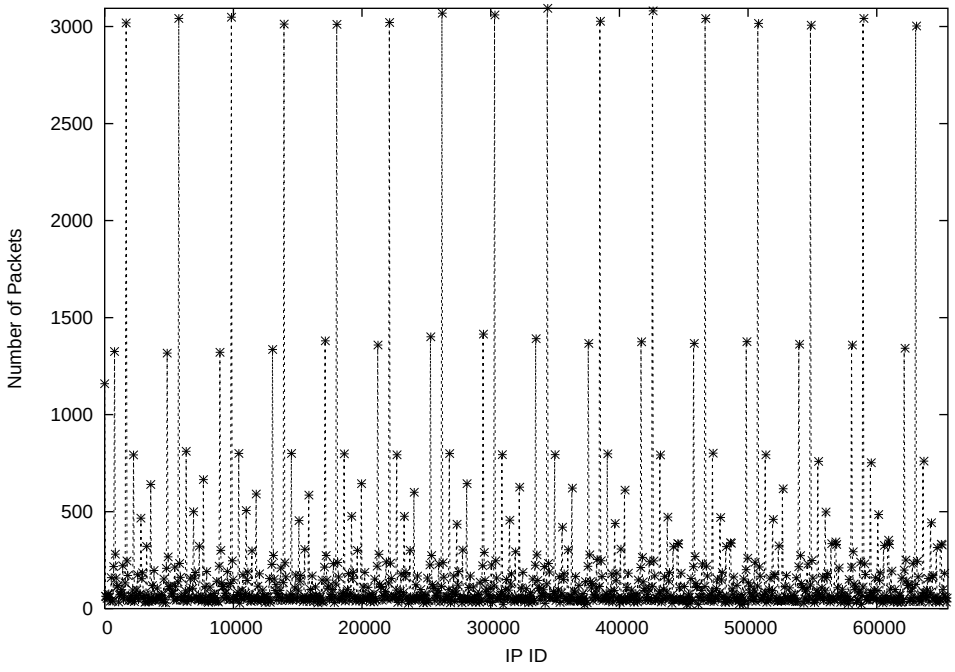
The long-term traces also present evidence of systematic sampling (Effect 1). Figure 3.8 (top) shows an histogram of the packet IP IDs for the traffic captured by RND. The distribution of IP IDs is uniform, with the exception of the IP ID = 0 point, due to the existence of diverse *ZERO* stacks. The bottom figure shows the same histogram, but for the corresponding SAMP trace.

We can see a strong bias in several values. Every vertical, standing-out "line" of values corresponds to a series of packets from the same (large) connection. The reason of having 16 equidistant lines is that our sampling rate is 1:4096, which in 65,536 values implies 16 sampled ones.

The same systematic sampling effect can be seen in a different way. Figure 3.9 shows the life of a given large connection (actually the biggest one in its trace). The top figure shows the connection as sampled by RND, while the bottom figure shows the same connection as sampled by SAMP.



(a) rnd



(b) samp

Figure 3.8: IP ID Distribution for the 2004/01/15 Trace

Every point in the trace represents a packet, with the x axis showing its IP ID, and the y axis representing its timestamp. The line stitching all the points together is strictly increasing in the y axis. In other words, it is a time line. We can see the strong systematic 1 in N effect in the SAMP-captured connection. Almost all packets in both graphs are 1500-byte long.

As an interesting side effect, the same systematic sampling bias can be seen in the reverse path. In this case, all reverse-path packets are 40-bytes long.

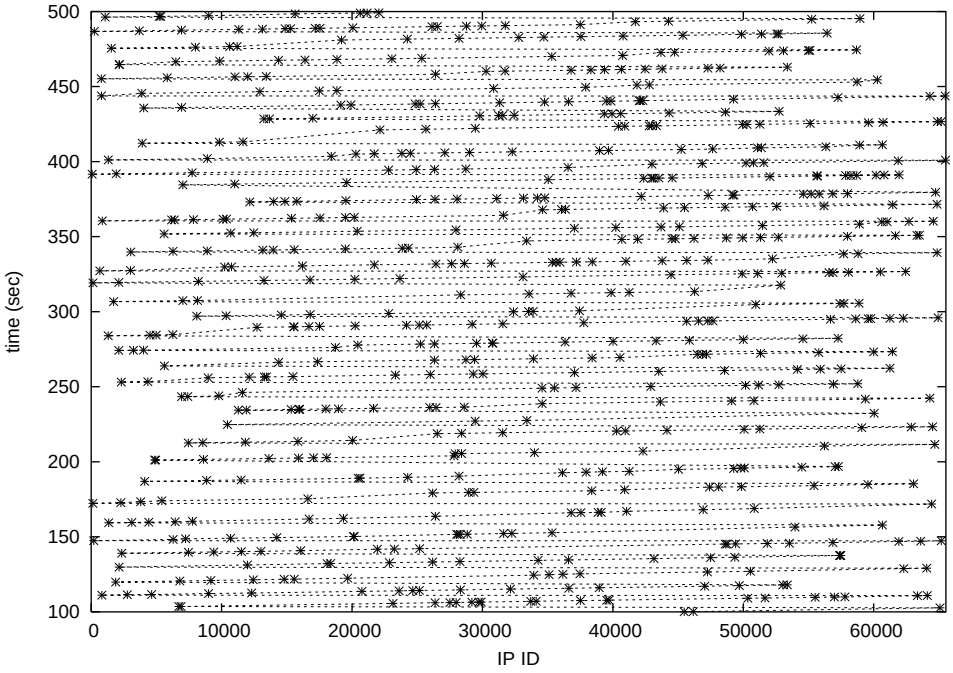
Figure 3.10 shows the same effect as Figure 3.9, but for the reverse path.

Finally, this aliasing produces aliasing in the time domain. We analyzed the interpacket timing for the same connection (Figures not shown). While the timing distribution in the case of packets captured by RND is centered very close to zero, the distribution in the case of packets captured by SAMP is bimodal, with one mode in zero and the other close to 0.5 seconds. This reflects the fact that 1 packet in 4096 is being systematically sampled, so therefore the connection bandwidth is close to 8192 packets per second.

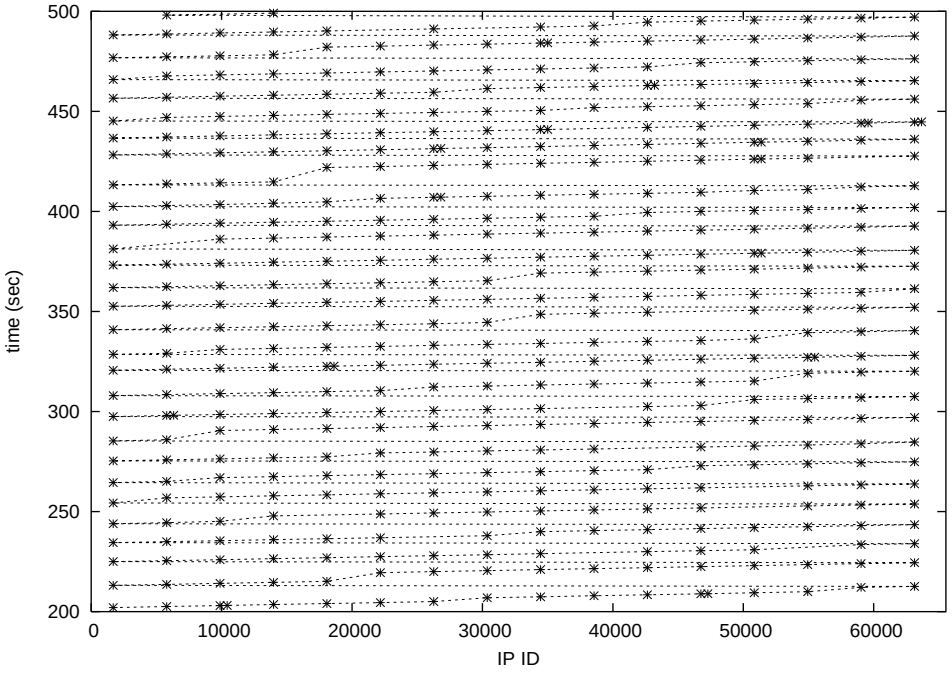
3.6.3 Conclusions

The main conclusions on comparing RND and SAMP or NIDZ are:

- Effect 1: Because of the prevalence of *CONSECUTIVE* stacks, most large connections are sampled systematically by SAMP, instead of randomly. Systematic sampling reduces the variance on the amount of traffic captured by SAMP,

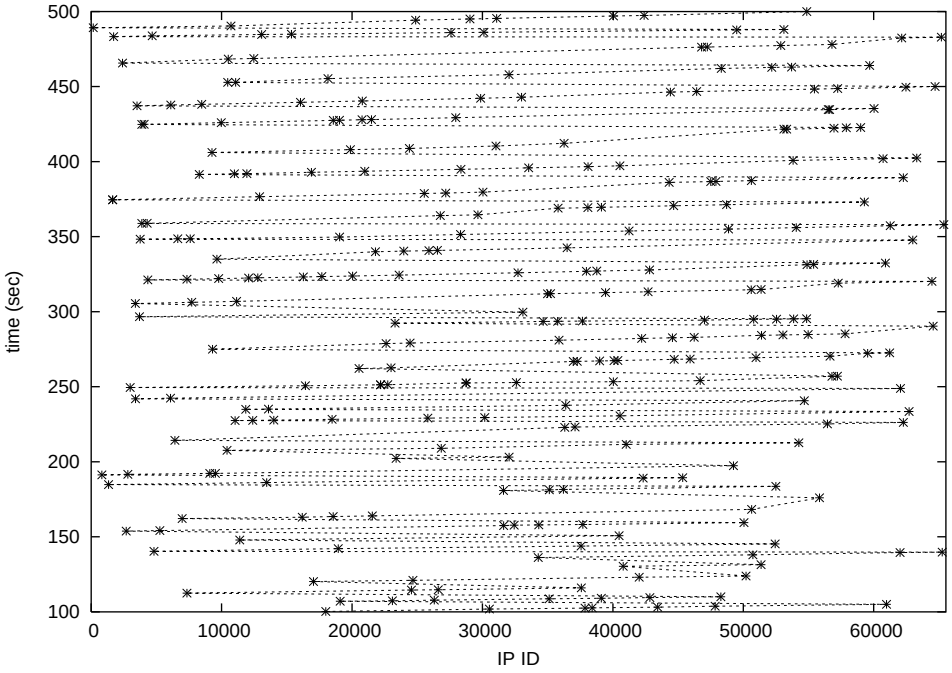


(a) rnd

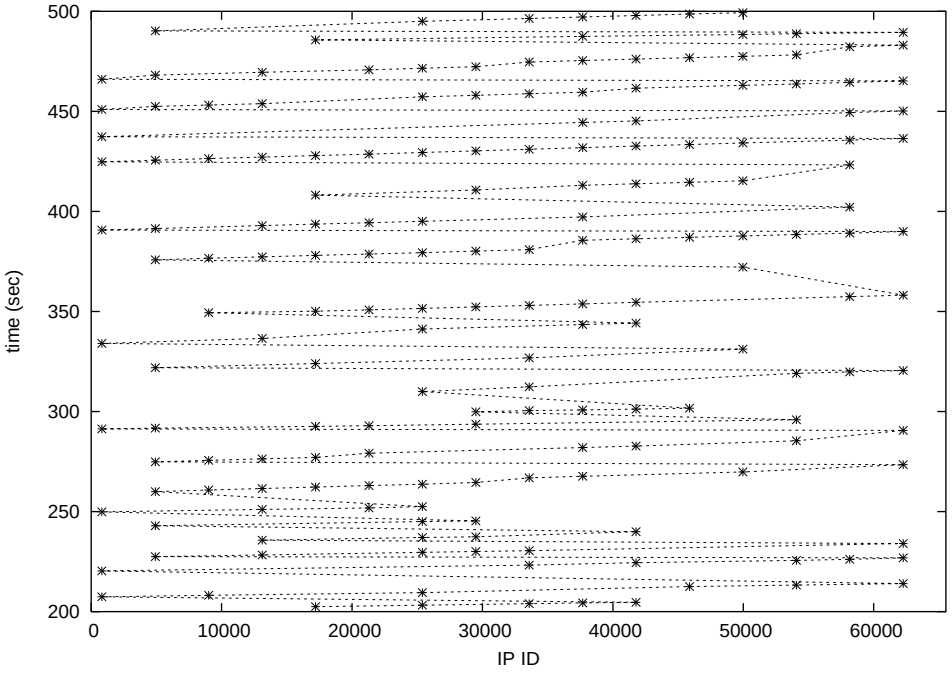


(b) samp

Figure 3.9: Timeline for Biggest Connection in 2004/01/15 Trace



(a) rnd



(b) samp

Figure 3.10: Timeline for Biggest Connection in 2004/01/15 Trace (Reverse Path)

introduces intertiming bias, and may present problems when the property being measured is periodic, and the sampling ratio is near its period [Lohr, 1999]. Systematic sampling is typically symmetric, affecting both directions of a flow.

- Effect 2: Because of the existence of *ZERO* stacks, some subsets of traffic are aliased by SAMP. This aliasing can be positive (all the packets in the subset are captured by SAMP), or negative (no packets in the subset are captured by SAMP).

  As the sampling ratios used are normally very small, negative aliasing is uncommon and its effect steady but not very important (SAMP captures slightly less traffic than what it should). Positive aliasing is extremely uncommon, but when it happens, its effect can change significantly the amount and profile of the captured traffic.

  Positive aliasing can be fixed by using NIDZ instead of SAMP.

  Aliasing may be asymmetric: One should use caution when drawing conclusions based on a single connection.

- Effect 3: Because of the selective use of *ZERO* stacks, sampling of some subsets of the traffic is biased. This includes at least TCP control segments and UDP traffic.

These three effects are not a risk, but a reality. From experiments in a real environment, we can conclude how well SAMP and NIDZ perform, and where they can

be trusted and where not.

The benefits of using SAMP and NIDZ are:

- The total amount of captured traffic by SAMP is very similar to RND, just 1% lower [All percentages refer to packets. Byte differences are typically even smaller.] The total amount of traffic captured by NIDZ is also very similar, just 2% lower.

- NIDZ (SAMP plus eliminating all packets with zero IP ID) does a good job of fixing positive aliasing problems. Negative aliasing problems cannot be fixed.

The drawbacks of using SAMP and NIDZ are:

- TCP SYN/ACK sampling (and to a small degree TCP FIN or RST sampling) is flawed. One should not draw conclusions based on TCP control segments.

- UDP sampling is flawed. One should not draw conclusions based on UDP traffic.

- Using NIDZ biases against some Operating Systems that set the IP ID to zero in certain environments.

- Packet inter-timing is flawed. (This is probably irrelevant in sampled traffic anyway.)

3.7 State Addition

The second modification consists of the introduction of state in the packet filter. We extend the packet filter pseudo-machine architecture to include a persistent memory unit, which permits storing and recovering information between different packets. We provide fine-grained access to this memory unit, using both BPF packet-filter programs, and the standard device control interface.

3.7.1 Related Work

The idea of adding some sort of persistent (inter-packet) state to BPF is not new. Our approach differs from previous work in the generality of the persistent-state management capabilities.

MPF and mmdump are ad-hoc solutions to specific problems. MPF uses persistent state to match fragments to flows, and therefore dispatch fragments only to interested applications [Yuhara et al., 1994]. mmdump uses persistent state to keep multimedia session statistics [van der Merwe et al., 2000].

xPF [Ioannidis et al., 2002] and FPL [Cristea and Bos, 2004], on the other hand, provide generic memory access (read and write). xPF keeps the BPF ISA, augmenting it. Its goal, nevertheless, is to provide a generic engine for executing monitoring applications, and therefore it provides a different, more generic high-level language to express filters. One of the effects of the added generality is that branch restrictions are eliminated, and therefore the filter execution time is not implicitly bounded anymore

by the filter program length.

FPL changes both the high- and low-level filter languages, instead using generic-purpose languages. Considering that the operation of packet filters consists of the kernel running user-provided code, FPL introduces security- and performance-related issues (Section 3.3.1).

3.7.2 Persistent State Addition

Our approach keeps both the BPF ISA and the high-level language (tcpdump expressions). We keep the same tcpdump expression language, whose programming model is easy to understand, popular, and at the same time well-suited for the packet filtering operation. We extend both languages, by adding primitives that permit access to the persistent memory.

Associative Arrays

The most common requirements for persistent memory in the context of packet filtering are the storage and retrieval of information associated with a subset of the packet fields. We believe these requirements can be fulfilled by providing a set of associative arrays, which can be accessed using a subset of the packet fields as the keys.

An associative array (also known as map, lookup table, or dictionary) is a data structure that associates keys with values. Operations available in associative arrays

include (a) lookup, which returns the value associated with a key, if present; (b) insert, which adds a {key,value} tuple to the table; and (c) delete, which erases the tuple associated with a key, if present.

Associative Array Usage Examples

Let's describe some examples of packet-filter programs taking advantage of associative arrays.

The first case is connection filtering. Applications may want to manage information on a per-connection basis. A common case is performing connection sampling, where the decision to sample a packet or not is the same for all packets belonging to the same connection.

Connection sampling can be implemented using an associative array whose key is the 5-tuple (104-bit) that defines a connection (source and destination addresses, transport-layer protocol, and source and destination ports), and whose value is a 1-bit long quantity that, when interpreted as a Boolean value, decides whether packets from such connections must be captured.

The operation of such implementation of connection sampling is simple. On receiving a packet, the packet filter checks the connection-sampling associative array. Then, it reads the bit value, and depending on the value, decides to filter in the packet or not.

Fragment filtering can also be implemented using associative arrays. The problem

with filtering fragments is that those that are not the first one in a fragment series do not carry transport headers. Therefore, any filtering that relies in transport header information cannot be applied to them.

A possible solution is to store the transport header of the first fragment in a series (or at least some of its fields, for example both the source and the destination port) in a fragment table, indexed by the source address, destination address, and IP ID field. When receiving the first fragment in a series, its source and destination port are stored in the table, indexed by the packet's source and destination address and IP ID field. When receiving a fragment different from the first one in a series (i.e., without transport header), the packet filter will get both ports by querying the table.

This table does not help in the case of out-of-order fragments. In this case, the filtering decision can be moved from the packet filter to the application itself.

Another example of the usage of associative arrays is IP address tables. These tables can be used, for example, to implement lists of interesting (whitelists) and uninteresting (blacklists) hosts in an efficient and dynamic fashion. On receiving a packet, the packet-filter checks the packet address in the whitelist and blacklist hosts table. If the host address is in the first table, the packet is captured. If the host address is in the second table, the packet is rejected.

Why Not Generic BPF?

Note that generic BPF has the required functionality to implement the three examples mentioned before. As an example, let us consider the address whitelist example. It is possible in BPF to implement a host whitelist by using as a filter the disjunction of several "(host = host_address)" primitives (one for every host in the list).

There are two problems with this approach, namely efficiency and dynamic access. The efficiency problem is due to the sequential nature of the operation of BPF filters. In order to run a plain BPF filter storing primitives for N host addresses, the BPF engine must run the N primitives in sequential order, until it finds a match or it finishes. Therefore, its running time is $O(N)$.

The dynamic access problem relates to the static nature of BPF filters. If an application needs to make any change to its filter (either add a new primitive or delete an existing one), it must create the new filter from scratch: write the tcpdump expression, compile and optimize it, and then send it to the kernel, so that the latter checks and installs it. This can take a long time when the number of primitives is large.

Both problems mean that BPF does not scale to more than a few hundreds of primitives. In comparison, looking up an address in a dictionary takes $O(1)$ time, and there is no compile cost to add or delete a new address.

Associative Array Requirements

The examples above introduce several requirements for the functionality of the associative arrays:

- arbitrary-length values: As shown in the examples, the values stored in the dictionary may have different sizes. In some cases, applications need only a decision on whether the packet will be accepted or not. This requires just 1 bit per value. In the fragment dispatching case, applications need two transport-layer ports, which require a 32-bit value.

- arbitrary-length keys: Keys used to query the dictionaries may also have different lengths. In our examples, we have seen cases from 32 bits, in the case of IP addresses, to 104 bits, in the case of per-connection filtering.

- multiplicity: Applications may need several dictionaries, with different key and value widths. A one-size-fits-all dictionary using the largest key and value widths would be too inefficient.

- arbitrary-length size: The kernel must keep state for all the dictionaries of a packet-filter application. Applications should be able to decide the size of their dictionaries.

3.7.3 Persistent State Design

Our extension to add state to BPF consists of including a number of set-associative hash tables. We add to every BPF device a set of chunks of memory, whose size is determined on initialization by the user, and which are accessible as hash tables through some basic primitives.

Design Space and Applicability

The design space of the hash tables presents several alternatives that influence the applicability of the persistent state addition to different scenarios.

The first issue is whether the size of the tables must be fully dynamic (growing with each new entry), fixed, or fixed but resizable. Dynamic-sized tables are definitely more flexible, but they introduce potential security hazards. For example, consider that the hash tables are used by a security monitoring application (e.g., a Network Intrusion Detection System, or *NIDS*), which creates state for every connection it sees. As the size of the memory allocated by the NIDS grows as a function of network traffic (the number of connection seen), this can be used by an attacker to increase the NIDS footprint, causing it to eventually crash. This is complicated by the fact that, in order to run efficiently, BPF must be run in privileged (kernel) mode. This means that an attacker could crash not only the NIDS, but also the host in which the NIDS runs.

An intermediate approach is to used fixed-size tables, but with the possibility of

on-demand resizing. This is, for example, the approach used by Bro [Paxson, 1999]. Bro dynamically resizes its internal hash tables when their hash bucket chains exceed a certain average length [Dreger et al., 2004]. Resizing a table requires to copy all pointers from the old table to the new one, which can take hundreds of milliseconds for large tables. Bro resizes its tables incrementally, keeping both tables for a while, and copying only a few pointers per new packet [Dreger et al., 2004].

The second issue is whether false false negatives are allowed. False negatives are by-products of the limitation in table sizes. If the hash table size is fixed, once it is full, the only way to deal with new insert requests is by evicting old entries. This may create false negatives, for example, when information on a given connection has been evicted from the tables because of capacity concerns.

For example, let's consider a table used to track suspicious connections. An attacker could take advantage of the table fixed size by creating multiple fake connections, increasing the table occupation until its has to evict entries. If she manages to cause the table to evict the entry corresponding to the connection she does not want to be tracked (the *culprit* connection), she is in effect hiding the culprit from monitoring.

The problem is made worse if finding which entries cause the eviction of a given entry is an easy task. This means that the number of fake connections the attacker needs to create in order to evict the culprit entry is the table associativity [Crosby and Wallach, 2003].

A better approach involves the use of strong hash functions in the table layout.

If it is not possible to know which entries will cause the culprit entry to be evicted, the only approach left for the attacker is brute force, i.e., adding a number of entries comparable to the table capacity, with the hope that the culprit one will be eventually evicted. Combined with a random eviction mechanism, this ensures the attacker is not able to deterministically control the contents of our tables.

The last issue refers to false positives. In order to maximize the use of the the space allocated for the tables, and especially if the key used is large (e.g., the 104 bits used by the traditional 5-tuple connection definition), there is the possibility of storing a hash of the key, instead of the full key. For example, instead of the 104 bits, we could store a 32 bit hash of it.

This works fine as long as the occupation of the original key space is sparse (which is the case, for example, of the 104 bit connection definition). If the hash function is strong enough, it ensures that the probability of 2 connection keys hashing to the same 32 bit value (a false positive, in which a connection is dealt with as if it were another) is marginal. Another disadvantage of storing the hash value is that we cannot know the real key of the table entries, just its hashed value.

Our approach is to use fixed-size tables, and to give the user full control over whether the original keys or a hashed version of them are stored as table entry keys. Note that, in any case, the interface (API) provided by the BPF state additions keeps the same. A possible addition is to add on-demand resizing. This could be easily added by relying in the current table size ioctl function (see Table 3.5).

The reason to choose hash tables between the different associative array structures available, and this particular design space point, is that the tables have $O(1)$ average lookup time, regardless of the number of objects in the table.[3] Moreover, this lookup time is not affected by resizing.

We expect that some of the tables will have a very large number of tuples, which makes average lookup time an important metric.

Hash tables provide 4 different access functions:

- "lookup: {table, key} → T/F": lookup a key in a given table.
- "retrieve: {table, key} → value": retrieve the value associated with a key in a given table.
- "insert: {table, key, value}": insert the {key,value} tuple in a given table.
- "delete: {table, key}": delete the tuple associated with a key in a given table.

In our packet-filtering scenario, we expect lookup to be the most common of the four operations.

Probabilistic Collision Resolution

One of the main issues when designing hash tables is collision resolution. In a hash table, a "collision" is defined as the case where the keys of two inserted {key,value} tuples hash to the same position in the table.

[3] In comparison, for example, self-balancing binary search trees have $O(\log n)$ average lookup time.

As a generic hash table does not know the keys in advance, perfect hashing is not possible, and collisions may occur. From the different approaches used to solve collisions, we have chosen probabilistic set-associative hash tables over chaining or open addressing.

Chaining (associating with each position a list of slots where tuples are stored) has two main drawbacks. First, it requires a memory allocator that is driven indirectly by a user application (traffic directed to the application), and that causes the lists to grow, potentially unbounded. This presents security concerns, as the tables are located in the kernel. Second, worst-case performance is $O(n)$ instead of $O(1)$. Note that this is an efficiency concern, but also a security one: Worst-case behavior may be due to either degenerated workloads or algorithm complexity attacks [Crosby and Wallach, 2003].

Open addressing (resolving collisions by setting a mechanism to look in alternate locations of the table for a given key) has two main efficiency drawbacks. First, deleting elements may be very costly, as it could require reorganizing the full table. Second, worst-case performance is also $O(n)$ instead of $O(1)$.

Probabilistic hashing resolves collisions by evicting tuples. In the simplest case, a given tuple can be located in just one position in the hash table. This means that two tuples $\{K_a,V_a\}$ and $\{K_b,V_b\}$ that collide cannot be in the table at the same time. If we want to insert $\{K_a,V_a\}$ in the table, an $\{K_b,V_b\}$ is already inserted, we must first evict the latter in order to make space for the former.

Probabilistic hashing does not require a memory allocator, has a fixed bound in the size of the stored data, and keeps an $O(1)$ worst-case behavior. It completely avoids the security concerns associated with unbounded table sizes by fixing the amount of memory used by each packet filter.

The main tradeoff of probabilistic hash tables is that they may produce false negatives. In the previous case, if we lookup the tuple $\{K_b,V_b\}$ after it has been evicted, it will not be found.

In order to limit the amount of evicted tuples, we introduce associativity into the hash table. In a w-way associative hash table, the table entries are clustered in groups of w consecutive positions. A tuple is inserted in the emptiest entry of the group to which it hashes (ties are broken arbitrarily).

Associativity decreases the probability that a collision causes an eviction, as the w entries in a group must be full before there is an eviction. On the other hand, the lookup function must check all the entries in every group, and therefore the lookup performance is $O(w)$ instead of $O(1)$.

Bloom Filters

The second issue is space use, and therefore relates only to efficiency. Hash tables used in packet filtering processes typically have keys wider than the values they index. This means most of the hash table data buffer must be used to keep the keys. For example, connection sampling tables require 104 bits of key space for every bit of

value space.

A related issue is the fact that the current BPF Virtual Machine registry is composed of only two 32-bit registers, which makes it very hard to implement operations that involve more than 32 bit items.

To address both issues, our solution proposes a more efficient approach, which saves space by keeping a hash value of the key instead of the full key. This is a special case of a Bloom filter [Bloom, 1970], where a) the number of hash functions is $k = 1$, and b) we use an array of 2^v different values (where v is the width of the values) instead of an array of bits. Note that the latter is possible because $k = 1$.

This data structure may result in false positives, i.e., returning a result when the queried key is not in the array. Consider two keys K_a and K_b, mapped to the values V_a and V_b respectively, which hash to the same hash value H_a. Consider also that the tuple $\{H_a, V_a\}$ is stored in the table. A lookup query for K_b will return V_a instead of V_b.

The operation of the final data structure is shown in Figure 3.11. The original tuple $\{K_a, V_a\}$ is transformed into a narrower tuple $\{H_a, V_a\}$, where $H_a = h_1(K_a)$. The reduced tuple is inserted into a probabilistic hash table using a second hash function, $h_2()$.

Note that, in order to avoid clustering effects in the hash table, it is enough for the first hash function ($h_1()$) to cause as few collisions as possible. For the second hash function, $h_2()$, we use a simple *mod* function.

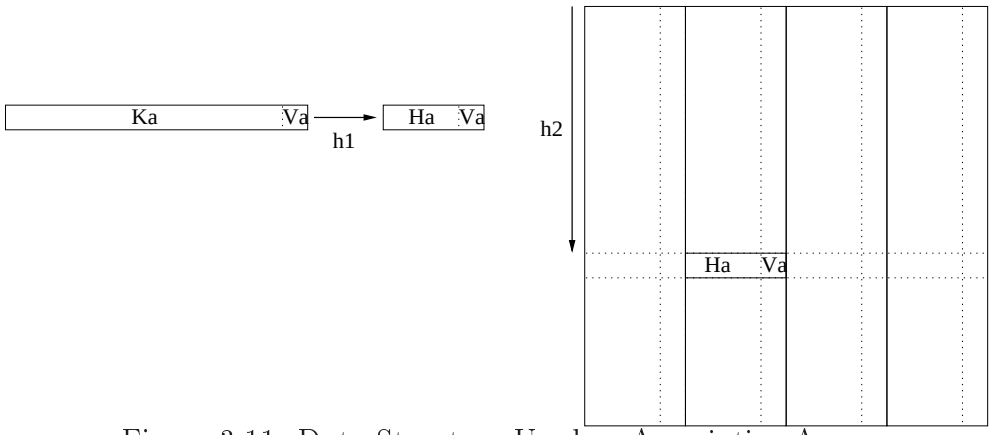


Figure 3.11: Data Structure Used as Associative Array

Hash Functions

For the first hash function, $h_1()$, we provide three options:

1. Linear Congruential Generator (LCG): LCG is a simple hash function. The reason for providing this hash function instead of permitting the application to build it by itself is the lack of the modulus operator in BPF. On the other hand, it is prone to to worst-case behavior with either degenerated workloads or algorithm complexity attacks [Crosby and Wallach, 2003].

2. Message Digest ($MD5$) [Rivest, 1992]: MD5 is a cryptographic hash function. It is slower than the LCG-based function, but it provides pseudo-random values.

3. Universal Hash Functions ($UHASH$) [Carter and Wegman, 1979]: Universal Hash Functions provide less strong guarantees than cryptographic hash functions, but are much faster.

Programming Model

We provide two methods to access to the memory unit, namely the standard device/socket control mechanism, and direct access via BPF packet-filter primitives.

The first method is fairly straight-forward. The user application makes requests to the kernel by calling the ioctl function in the BPF device descriptor (setsockopt in the socket filter when the Operating System uses the socket API to implement BPF, as in the Linux Socket Filter case). The kernel captures such requests, and honors them.

Tables 3.4 and 3.5 show the ioctl list for configuring and accessing the hash functions and the hash tables, respectively. (Note that BIOCSHTSIZET can only be used at initialization time.)

command	explanation
BIOCSHFLCGSEED	set the LCG hash function seed
BIOCGHFLCGSEED	get the LCG hash function seed
BIOCGHFLCG	get the LCG hash value of a key
BIOCSHFMD5SEED	set the MD5 hash function seed
BIOCGHFMD5SEED	get the MD5 hash function seed
BIOCGHFMD5	get the MD5 hash value of a key

Table 3.4: ioctl API to the Hash Functions

The second method permits direct access to the hash functions and tables from the BPF program itself.

To access the hash functions, we provide two new tcpdump primitives, namely hash_lcg and hash_md5. Both of them accept a variable number of arguments, and return the hashed value of the concatenation of all the arguments. The first one uses

the LCG hash function, and the second one uses the MD5 hash function.

To access the tables, we provide four new tcpdump primitives, namely lookup, retrieve, insert, and delete. They implement the hash tables functions of the same names.

The two query-only primitives are relatively easy to integrate in a complex packet filter. For example, the following filter can be used to perform TCP connection sampling.

(lookup(0, hash_lcg(ip[12:4], ip[16:4], tcp[0:2], tcp[2:2]))) or

(lookup(0, hash_lcg(ip[16:4], ip[12:4], tcp[2:2], tcp[0:2])))

This filter calculates the hashed value of the key formed by the 104-bit TCP

command	explanation
BIOCSHTNUMBER	set the number of tables
BIOCGHTNUMBER	get the number of tables
BIOCSHTID	set the working hash table
BIOCGHTID	get the working hash table
BIOCSHTSIZET	set the table size (bytes)
BIOCGHTSIZET	get the table size (bytes)
BIOCSHTSIZEA	set the associativity
BIOCGHTSIZEA	get the associativity
BIOCSHTSIZEK	set the key size (bits)
BIOCGHTSIZEK	get the key size (bits)
BIOCSHTSIZEV	set the value size (bits)
BIOCGHTSIZEV	get the value size (bits)
BIOCSHTDEFAULT	set default value when non-existent entry
BIOCGHTDEFAULT	get default value when non-existent entry
BIOCSHTPUT	insert a {hashed key,value} tuple
BIOCGHTGET	lookup a hashed key, and return the corresponding value
BIOCSHTREM	mark the entry associated with a hashed key as invalid

Table 3.5: ioctl API to the Hash Tables

connection tuple (note that the IP protocol in this case is set to the TCP number), and uses it as the key to lookup the corresponding value in table 0.

The other two primitives (insert and delete) are harder to integrate in useful filters: they modify the tables, and typically they must be run only in a small percentage of the packet filter runs, when some condition holds. For example, in connection sampling, an insert operation may only be carried out when the first packet of a new connection is seen. Conversely, a delete operation will only be carried out when the last packet of a new connection is seen.

Note that this works provided that tcpdump expressions are evaluated left-to-right. This may not be always the case, as the BPF optimizer that translates the high level filters into BPF programs has complete freedom to reorder the former [McCanne and Jacobson, 1993]. Our approach depends therefore on limiting the reordering flexibility of the optimizer.

Once primitive reordering has been forbidden, tcpdump expressions provide a limited form of flow control by considering the left-to-right evaluation order of filters. The way to achieve conditional evaluation of a primitive is by setting a filter composed of the conjunction of the condition and table access primitives, in this exact order. If the condition primitive is not true, the conjunction is also false, and therefore the second primitive need not be evaluated. If the condition is true, the conjunction is true or false depending on the second primitive (the state-modifying primitive).

As an example, the following filter performs random connection sampling, so that

all the packets in one in four connections are captured.

```
(lookup(0, hash_lcg(ip[12:4], ip[16:4], tcp[0:2], tcp[2:2]))) or
(lookup(0, hash_lcg(ip[16:4], ip[12:4], tcp[2:2], tcp[0:2]))) or
( (tcp[13] & 18 = 2) and
        (random(4) = 1) and
        (insert(0, hash_lcg(ip[12:4], ip[16:4], tcp[0:2], tcp[2:2]),1)) )
```

The first two primitives check whether the connection tuple is already in the connection table. The last primitive decides whether packets from a connection are to be sampled or not. It is composed of three sub-primitives, from which the first one checks for SYN segments (beginning of a connection). If it does not hold, the full primitive is false, independently of the other two sub-primitives, and therefore, the latter are not evaluated. In the same manner, the second sub-primitive takes a random decision, which is true one in four times. If the first and second sub-primitives are true, then the third one is evaluated, and the tuple information is stored in table 0.

As another example, the following filter counts TCP per-connection bytes.

```
1 (lookup(0, hash_lcg(ip[12:4], ip[16:4], tcp[0:2], tcp[2:2])) and
2         insert(0, hash_lcg(ip[12:4], ip[16:4], tcp[0:2], tcp[2:2]),
3                 ip[2:2] +
4                 retrieve(0, hash_lcg(ip[12:4], ip[16:4], tcp[0:2], tcp[2:2])))) or
```

```
5 (lookup(0, hash_lcg(ip[16:4], ip[12:4], tcp[2:2], tcp[0:2])) and
6         insert(0, hash_lcg(ip[16:4], ip[12:4], tcp[2:2], tcp[0:2]),
7                 ip[2:2] +
8                 retrieve(0, hash_lcg(ip[16:4], ip[12:4], tcp[2:2], tcp[0:2])))) or
9 insert(0, hash_lcg(ip[12:4], ip[16:4], tcp[0:2], tcp[2:2]), ip[2:2])
```

While looking slightly more complicated than the previous ones, the filter is easy to understand: Lines 1-4 check for the connection in the forward direction (lookup primitive). If the connection is found there, the current length in bytes is retrieved from the table (retrieve primitive), added to the current packet size (ip[2:2] primitive), and then reinserted in the table (insert primitive). Lines 5-8 do the same process but in the backwards direction. Line 9 is run if none of the previous primitives are true. It creates a new entry for the packet's connection, and inserts the size of the packet.[4]

Finally, we want to remark that this filter can be installed in the kernel, and run without the need for kernel boundary crossings.

3.7.4 Implementation of Hash Access Using BPF Primitives

Hash Functions

The implementation of the hash functions is relatively straightforward. The main difficulty is that both hash_lcg and hash_md5 may have a variable number of

[4]We are interested in adding more idioms or keywords in order to make for easier filter construction. For example, we could use hash_tcp_fwd to obtain hash_lcg(ip[12:4], ip[16:4], tcp[0:2], tcp[2:2]).

arguments. We add 3 new ALU operation modes to the BPF ISA, namely BPF_HBEG, BPF_HUPD, and BPF_HEND.

A BPF_HBEG ALU operation is generated at the beginning of a hash function. This operation stores the seed in the A register (for the LCG case), or resets an MD5 internal buffer (for the MD5 hash case).

A BPF_HUPD operation is generated for every argument in the hash function, except the last one. In the LCG hash case, this operation stores into the register A the result of applying the LCG operation to the bitwise exclusive OR of A and the value of the argument. In the MD5 case, it just appends the new argument to the MD5 internal buffer.

A BPF_HEND operation is generated for the last argument in the hash operation. In the LCG case, the result is the same as the previous operation. In the MD5 operation, the last argument is appended to the MD5 internal buffer, and then the MD5 hash operation is carried out. The first 32 bits of the result are copied into the register A.

Hash operations are not re-entrant.

Hash Tables

The implementation of the hash tables relies in adding BPF_HASH, a new addressing mode to the load (BPF_LD) and store (BPF_ST) operations. With the new addressing mode, the current hash table is accessed using register X as the

hashed key, and, in the case of the insert operation, register A as the tuple value.

3.7.5 Results

Performance

We are interested in comparing the performance of running a filter using our hash table approach to running the same filter implemented using plain BPF.

The trace used for the experiments is composed of 3 M packets, adding up to 345 MB. It consists of generic traffic generated by a small set of desktops and laptops (half of the traffic is HTTP) at the International Computer Science Institute (ICSI).

Figure 3.12 compares the performance of stateful BPF to (original) BPF. In the experiment, we have set both BPF and stateful BPF to capture only those packets whose source or destination address are in a given list (whitelist). The number of hosts in the list is shown in the x axis. The time taken to process each packet is shown in the y axis. The error bars in the BPF case show 95% confidence intervals.

The whitelist implementation in the BPF case consists of setting one primitive of the form "or host hostname" for each host. The implementation in the stateful BPF case uses one of the hash tables. The average time to access one element in the hash table is 1.827 us.

If the whitelist contains 12 hosts or more, the stateful BPF approach is faster than original BPF. In operational environments, blacklists with more than hundreds or thousands of addresses are not uncommon.

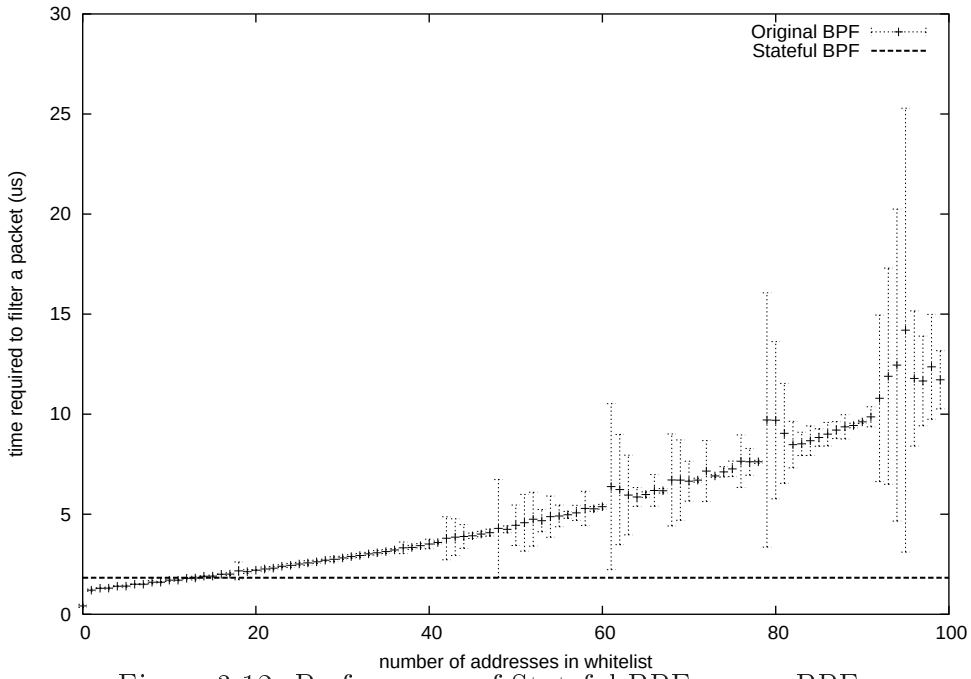


Figure 3.12: Performance of Stateful BPF versus BPF

Figure 3.13 shows the time required to compile a whitelist filter in the BPF case. Compiling a full filter is required every time a new host is added to or deleted from the list. The cost does not exist in the stateful BPF case.

Associativity

We are interested in understanding and measuring the influence of associativity in the probability of false negatives.

Consider a table composed of N entries. In this table, we insert m items (tuples), in the order they arrive. When an item arrives, the entry where it will be inserted is decided randomly. If another item was already occupying the frame, it is evicted. This is the traditional "balls-and-bins" problem.

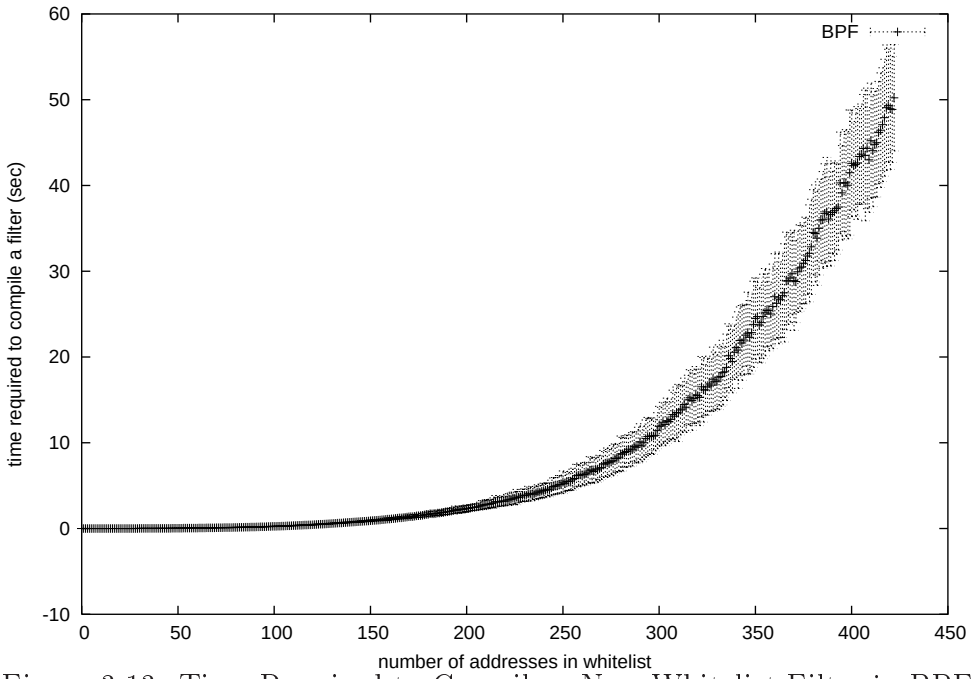


Figure 3.13: Time Required to Compile a New Whitelist Filter in BPF

In order to add associativity to the problem, we include a third parameter to the problem, w, which is the associativity of the table. The operation described above corresponds to the case $w = 1$.

When $w > 1$, the N frames in the table are associated in N/w groups of size w each. When an item arrives, one of the groups of frames is chosen randomly.

We want to know, for a given N, m, and w, a) the average number of evictions, and b) the probability that there will be an eviction.

We know that, for a given N, m, and w, we have N/w groups. Consider the jth group. We define the random variable $\mathcal{X}_j$ to be the number of items in frame j after the experiment. We know that $\mathcal{X}_j \sim Bin(m, p)$, where $p = w/N$.

We assume that the distribution of the number of items in a given group is $\mathcal{X}_j \sim$

$Poisson(\lambda)$, where $\lambda = mp = mw/N$. This way we can treat the group loads as independent, identically-distributed (IID) random variables.[5]

We know that $E[\mathcal{X}_j] = \lambda$.

We define the random variable $\mathcal{N}_j$ to be the number of evictions in $\mathcal{X}_j$. We know that:

$$\mathcal{N}_j = \begin{cases} 0 & \text{if } \mathcal{X}_j \leq w \\ 1 & \text{if } \mathcal{X}_j = w+1 \\ 2 & \text{if } \mathcal{X}_j = w+2 \\ & \cdots \\ m-w & \text{if } \mathcal{X}_j = m \end{cases}$$

The number of items evicted in the jth frame is $\mathcal{N}_j = \mathcal{X}_j - w$, except:

- when $\mathcal{X}_j = 0$, $\mathcal{N}_j = (\mathcal{X}_j - w) + w$,
- when $\mathcal{X}_j = 1$, $\mathcal{N}_j = (\mathcal{X}_j - w) + w - 1$,
- when $\mathcal{X}_j = 2$, $\mathcal{N}_j = (\mathcal{X}_j - w) + w - 2$,
- $\cdots$
- when $\mathcal{X}_j = w - 1$, $\mathcal{N}_j = (\mathcal{X}_j - w) + 1$,

[5]Note that $\mathcal{X}_j$ are definitely not IID. For example, if we know the value of all the random variables but the last one, the last one is completely determined. [Mitzenmacher and Upfal, 2005] discusses the applicability of this approximation.

Therefore, the average number evicted items in the jth frame is:

$$\begin{aligned} E[\mathcal{N}_j] &= (E[\mathcal{X}_j] - w) + wP(\mathcal{X}_j = 0) + (w-1)P(\mathcal{X}_j = 1) + (w-2)P(\mathcal{X}_j = 2) + \cdots + \\ &\quad +1P(\mathcal{X}_j = w-1) \\ &= \lambda - w + \sum_{i=0}^{w}(w-i)P(\mathcal{X}_j = i) = \lambda - w + e^{\lambda}\sum_{i=0}^{w}(w-i)\frac{\lambda^i}{i!} \end{aligned}$$

We define the random variable $\mathcal{M}$ to be the total number of items evicted. We know that $\mathcal{M} = \sum_{j=0}^{N/w}\mathcal{N}_j$, and therefore:

$$E[\mathcal{N}_j] = \frac{N}{w}\lambda - n + e^{\lambda}\sum_{i=0}^{w}(w-i)\frac{\lambda^i}{i!}$$

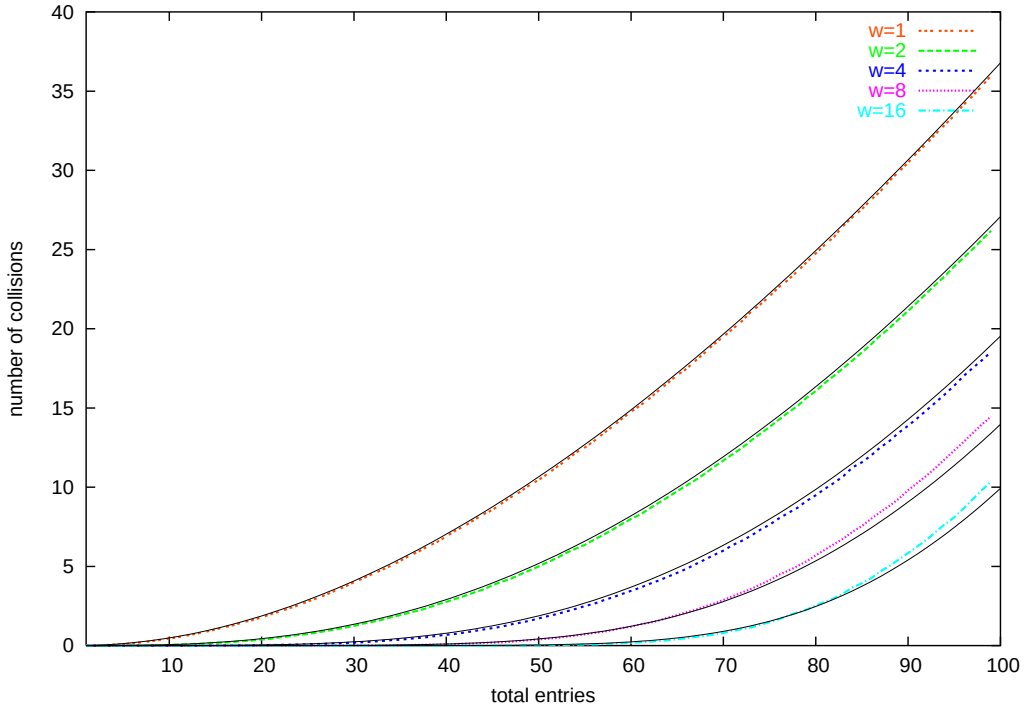
Table 3.6 shows some values of $E[\mathcal{M}]$ for small values of w.

w	$E[\mathcal{M}]$
1	$\lambda - 1 + e^{\lambda}$
2	$\lambda - 2 + e^{\lambda}(2+\lambda)$
4	$\lambda - 4 + e^{\lambda}(4 + 3\lambda + 2\lambda^2/2 + \lambda^3/6)$

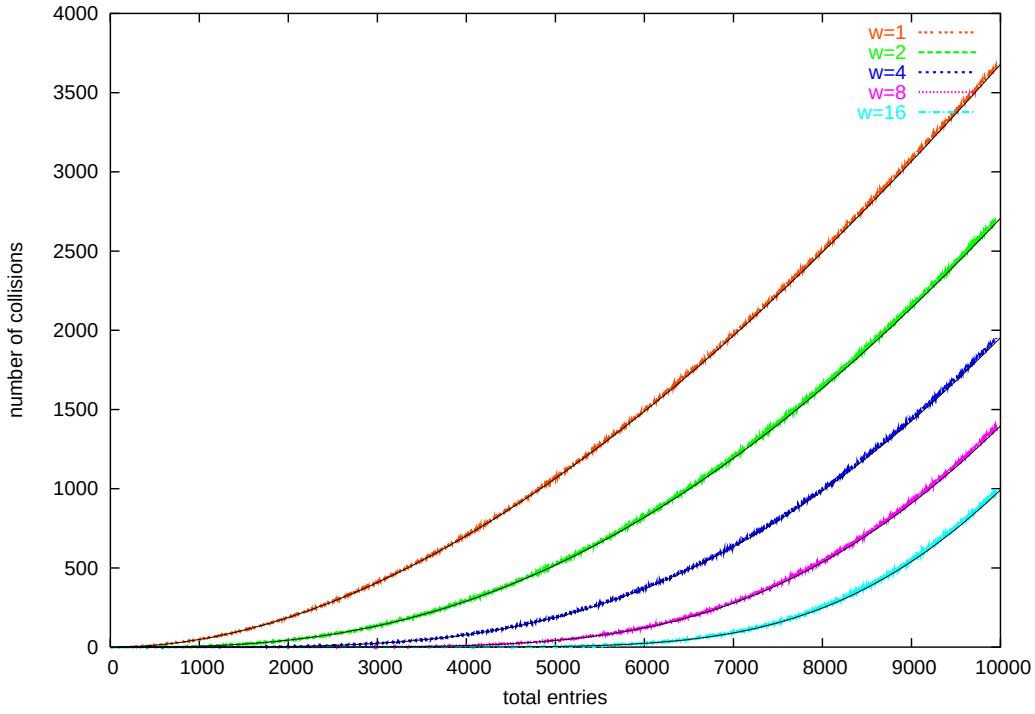
Table 3.6: Average Number of Evictions for Small Values of w

Figure 3.14 shows the number of evictions for a table with $N = 100$ and $N = 10000$ entries, for different values of w and m. The solid lines correspond to the theoretical results. The non-solid lines show the experimental results.

If the table is lightly loaded, we are also interested in the value of the table



(a) $N = 100$



(c) $N = 10000$

Figure 3.14: Theoretical and Experimental Number of Evictions

occupancy (m/N) when evictions start happening.

It can be shown that:

$$P(\text{no eviction}) \quad = \quad \left[e^{-\lambda} \sum_{0}^{w} \frac{\lambda^w}{w!} \right]^{N/w}$$

The average number of tuples required to start having collisions corresponds to the median $\bar{m}$, i.e., the value of m that causes $P(\text{no eviction}) = 1/2$. Unfortunately, the expression is not easily simplified. We have instead run some experiments for some values of N, m, and w.

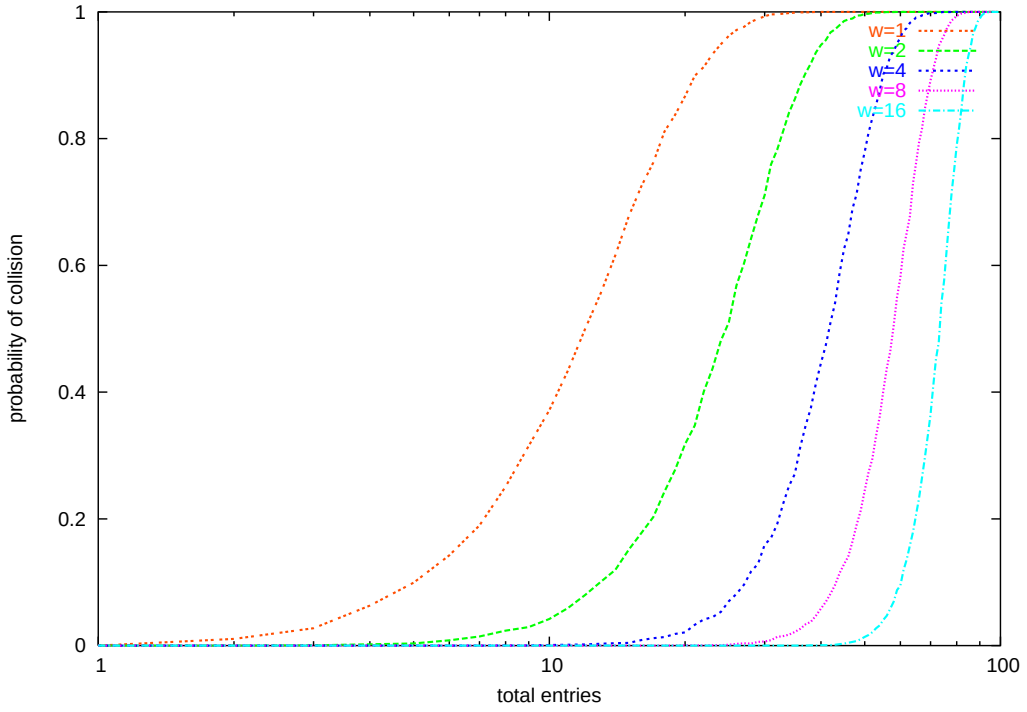
Figure 3.15 shows the experimental probability of at least an eviction happen, for a table with $N = 100$ and $N = 10000$ entries, for different values of w and m.

3.7.6 Applications

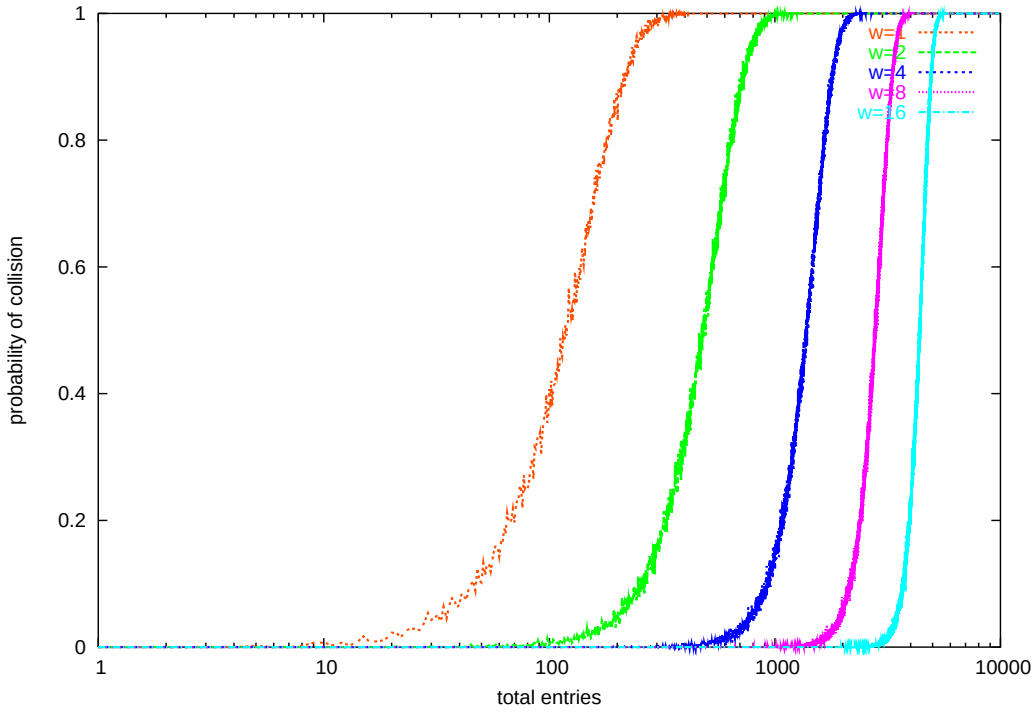
We envision several uses for this extension. First, we can implement Shunting without the need of a hardware device (see Chapter 4).

Second, we are interested in providing the ability of whitelisting and blacklisting hosts in very large numbers (thousands) which, as we have seen in Section 3.7.2, is not possible to do efficiently in current BPF.

Another main use is TCP connection sampling. We want to provide applications with the ability to randomly sample TCP connections, i.e., to sample all packets from



(a) $N = 100$



(c) $N = 10000$

Figure 3.15: Experimental Probability of an Entry Eviction

a random sample of connections.

Some researchers propose to perform connection or IP-pair sampling by relying only in plain BPF capabilities [Dreger, 2004]. The idea is to use as filter something like "(ip[12:4] xor ip[16:4] xor tcp[0:4]) mod P = R".[6]

[Moore et al., 2003a] describes a similar idea in a different scenario: load-balancing traffic between several network monitors. The authors propose (a) to use only the source and destination IP addresses, and (b) to hash the result of the xor using a P-valued hash function.

This filter concatenates the 104-bit connection using the xor operation (the IP protocol is set to the TCP number), calculates the modulus with a prime number (P in this case), and then picks one of the residues (R). The prime number is the inverse of the sampling ratio.

The main appeal of this approach is simplicity. As in the case of the SAMP approach to packet random sampling, it can be run in any BPF-based architecture.

The main drawbacks of this approach are, first, static behavior: Applications must decide which connections will be sampled when installing the filter. Second, a lack of flexibility: Applications cannot pick specific connections to be sampled. Third, potential existence of bias: The sampling is neither random nor uniform. The residue decides which connections will be filtered in, so this is actually deterministic sampling.

Moreover, [Dreger, 2004] does not deal with fragments, while [Moore et al., 2003a]

[6]Note that this is the logical filter. The real filter is slightly more complicated, as BPF has no xor, mod, or bitwise negation operations, and as connection sampling must include packets going in both forward and backward directions.

presents a strong aliasing, as all connections between two hosts are sampled the same way.

Other uses for the stateful approach include:

- In-kernel count of bytes per connection (already suggested).
- Capture of traffic from protocols that use control and data connections. A traditional operation mode in this type of protocol creates a control connection, which is specified by selecting a standard port, and one or more data connections, which are negotiated dynamically in the control connection. Therefore, data connections cannot be specified in advance. This is the case, for example, for FTP [Postel and Reynolds, 1985] and some multimedia protocols [van der Merwe et al., 2000].

  The case-example is FTP transmissions. If an application is interested in explicitly capturing FTP data connections, it may do so by capturing all packets corresponding to the standard FTP control port, or whose connection has been inserted into a hash table.

  The application requests and analyzes the control connection contents. As soon as it gets an FTP control packet that specifies an FTP data connection, it adds this connection to the hash table. Packets from the data connection will subsequently be captured.

  Note that this mechanism may create race conditions between recognizing the

negotiation and modifying the hash table.

Note also that this mechanism can also be applied to explicitly reject data connections. This can be useful for operational purposes. An example is monitoring systems focused on control connections where receiving the contents of data connections may impose an overwhelming load on the application.

- Dynamic filter control. We can use generic tables to use information that will be read for each filter run. The contents of such tables are updated by the user, which produces an arbitrary state BPF.

  An example of dynamic filter control is dynamic random sampling. Consider the case of an application that is randomly sampling 1 in 5 packets, using the primitive "random(5) = 0". At some moment, the application decides that it is too busy, and that it wants to move to sampling 1 in 50 packets. We want the application to be able to change the filter without flushing the state.

  This can be implemented by setting one of the hash tables with space only for one tuple with a 1-bit key and a 32-bit tuple. The sampling primitive will be "random(retrieve(id, 0)) = 0". This primitive will retrieve the value in position zero at the hash table with identifier id, and use it to calculate a random number between 0 and the value. We initialize the table by inserting the tuple {0, 5}. If we want to change the sampling ratio to 1 in 50, we just need to insert the tuple {0, 50} in the same table.

3.7.7 Future Work

In order to reduce the number of collisions and the probability that a given amount of tuples cause a collision, we are interested in adding 2-choice hashing for the management of hash tables. 2-choice hashing [Azar et al., 2000; Karp et al., 1992] consists of inserting tuples in a table by hashing with 2 hash functions, instead of only 1. The tuple is put in the emptiest entry in the table (ties are broken arbitrarily).

The main advantage of 2-choice hashing is that the tuples are distributed more evenly among the table entries. In fact, the number of tuples in the fullest entry in a 2-choice hash table has exponentially less tuples than the fullest entry in a normal hash table with the same capacity [Azar et al., 2000].

Adding more than 2 choices to the algorithm improves the effect of 2-choice hashing by only a constant factor [Azar et al., 2000]. Therefore, we will not consider it.

The main drawback of 2-choice hashing is that lookups get penalized in two ways. First, 2 hash values must be calculated. Second, 2 positions in the hash table must be checked, instead of one. The second penalty should be comparable to that in 2-way set-associative hash tables.

We have run some experiments for the number of evictions and probability of the first entry eviction for some combinations of set-associative hashing and 2-choice hashing.

Figure 3.16 shows the number of evictions for a table with $N = 100$ and $N = 10000$

entries, for different values of w and m, and for both 1- (i.e., normal 1-function hashing) and 2-choice hashing. Note that lines are laydown top-to-bottom, so the upper line is the one with more collisions.

Figure 3.17 shows the experimental probability of at least an eviction happen, for a table with $N = 100$ and $N = 10000$ entries, for different values of w and m, and 1- and 2-choice hashing.

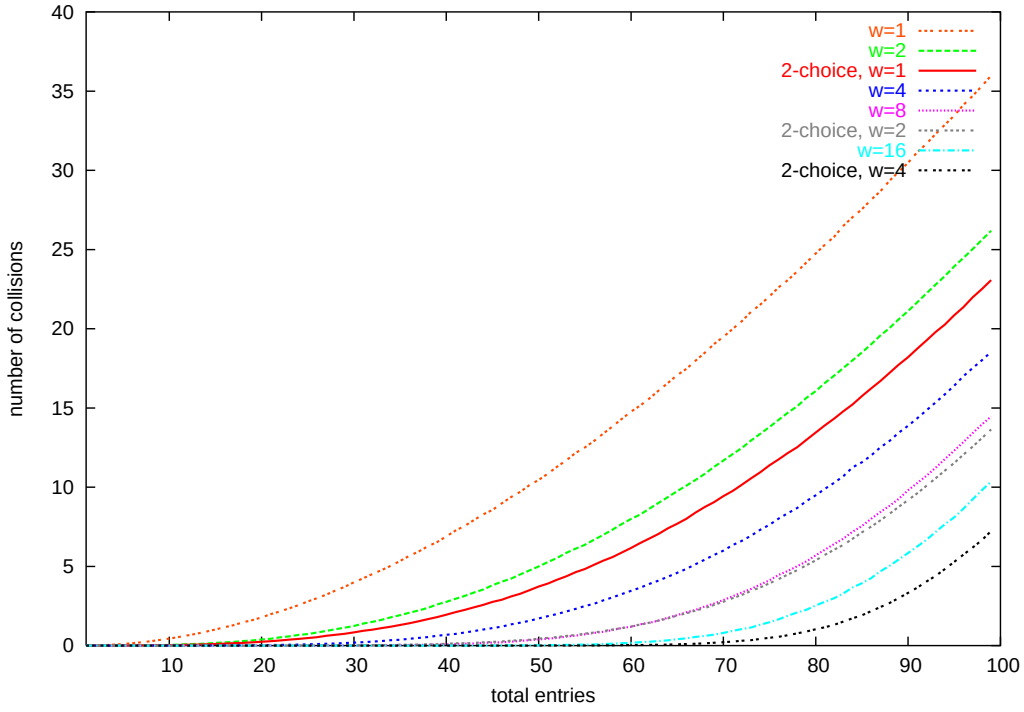
The graphics show 2-choice hashing as a promising alternative: The effect of 2-choice, 2-way set-associative hashing is similar to that of 8-way set-associative hashing. The cost is running 2 hash functions, and checking 4 entries in the 2-choice hash table. Compare to checking 8 entries in the 8-way set-associative hash table.

3.8 Summary

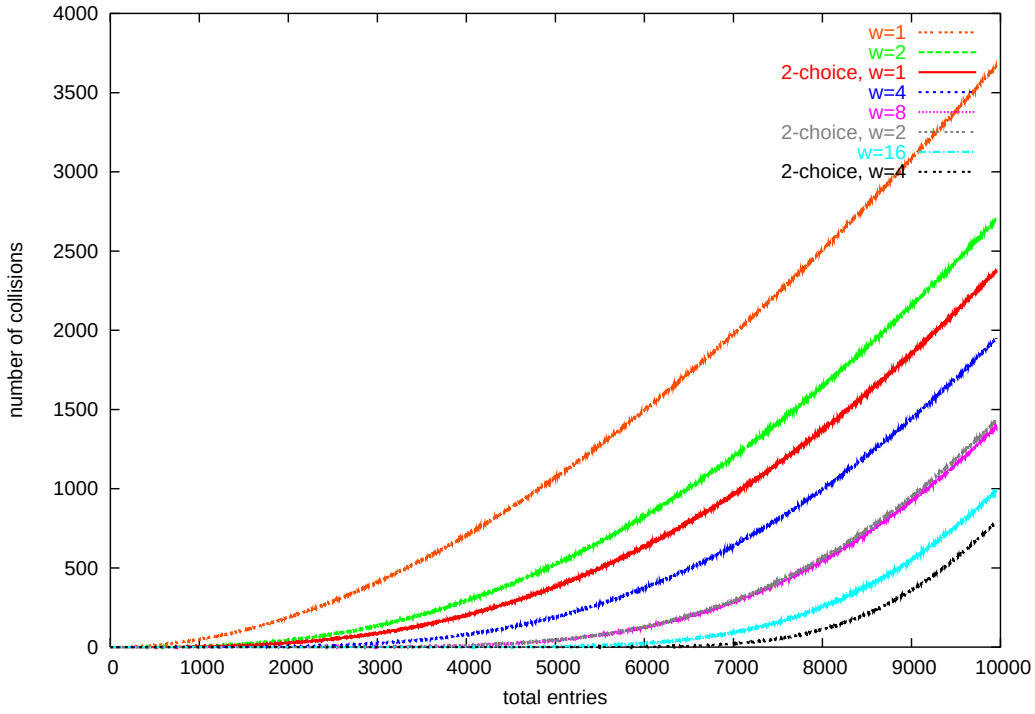
Current stateless filtering abstractions are very limited for the purpose of both packet- and connection-based sampling. This is most unfortunate in an IDS scenario.

We have developed two new mechanisms, random packet-sampling and state management, to the popular packet capture library (*libpcap*) running on Berkeley Packet Filter (*BPF*)-based machines [McCanne and Jacobson, 1993]. The main goal is to keep the simplicity (and therefore the performance) of traditional packet filters while increasing its flexibility.

The first addition is random sampling. While the implementation is extremely simple, we want to understand how other approaches that simulate random sampling

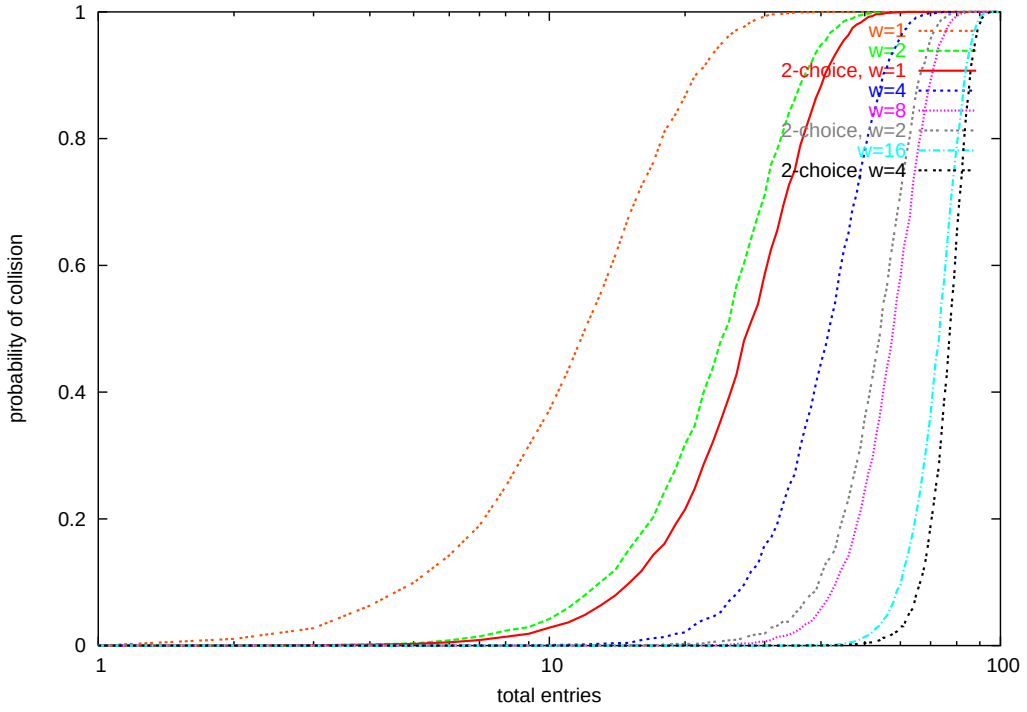


(a) $N = 100$

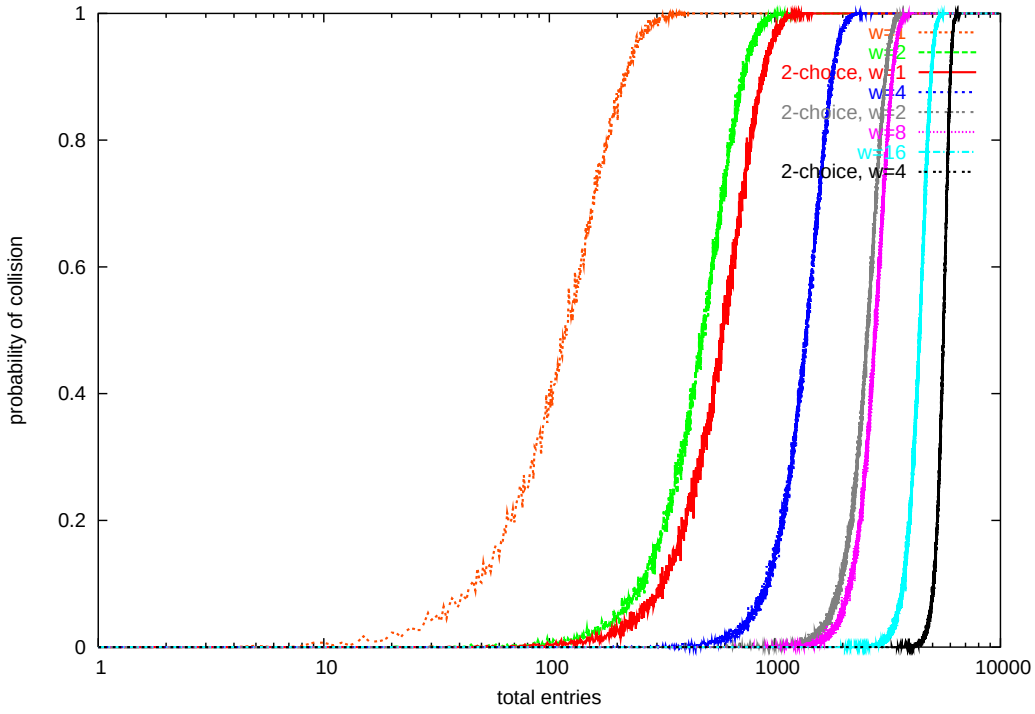


(c) $N = 10000$

Figure 3.16: Experimental Number of Evictions



(a) $N = 100$



(c) $N = 10000$

Figure 3.17: Experimental Probability of an Entry Eviction

perform compared to true pseudo-random sampling. A common approach is to assume uniform randomness in IP checksum fields, and sample packets by masking these fields. We have run some experiments to evaluate the strengths and weaknesses of this sampling method.

We have found that, after solving a main weakness, it works fine in most scenarios. This weakness is the source of randomness in IP headers. The main source is the IP ID field. This means connections where one of the end hosts uses a zero IP ID (e.g., some Linux hosts) are normally aliased (either positively or negatively). Even after filtering out those hosts, we have still found some aliasing effects in the composition of the sampled traffic.

The second addition is inter-packet persistent state. We provide a set of probabilistic, Bloom-filter based, set-associative hash tables that permit efficient access to a fixed chunk of memory. This access can be carried out through the standard device/socket control mechanism, or directly using the BPF filter.

The main advantages of the state addition programming model are, first, simplicity: We keep the tcpdump expression model, and just add dictionaries for inter-packet state. This keeps the implementation simple, and therefore we do not need to resort to eliminating branch restrictions. Second, space efficiency: This comes at the cost of a small probability of both false positives and false negatives. Third, performance efficiency: Our modifications permit stateful packet processing (for example, random connection sampling) without ever crossing the kernel-user boundary.

Both approaches introduce a series of new filter mechanisms, which provide richer, fine-grained control and new abstractions to the filtering process.

As an example of the power of these abstractions, we show a secure and efficient packet processing mechanism to count per-connection bytes that never sends a packet from kernel space to user space.

"Beware the Ides of March."

— WILLIAM SHAKESPEARE, Julius Cesar (I, ii, 33) (1599)

Chapter 4

Shunting

4.1 Abstract

This Chapter describes and motivates a novel architecture that permits high-speed, extensive (non-sampled and in-depth), stateful, inline traffic processing by integrating a simple, active, hardware device with a complex, software, decision engine. The basic idea is that the hardware device provides simple mechanisms to process packets, which are leveraged by the software decision engine to offload work into it. By taking advantage of the heavy-tailed nature of the processed traffic, a limited-size hardware device can process most of the packets, which never reach the software decision engine. This way, the software decision engine task gets limited to performing per-packet analysis on just a small subset of the traffic, and to setting the policy that drives the hardware device mechanisms.

We present an application of the Shunting technique to monitoring network intrusion. Our implementation permits full, in-depth network intrusion detection and prevention in Gigabit links. We use a modified version of a Network Intrusion Detection System as the basis of the software decision engine. We provide an evaluation of the behavior and performance of our implementation. We also describe the operational experience resulting from running such implementation in a real environment.

While we use the Shunting architecture to carry out intrusion monitoring, we argue that its usefulness is not limited to network security. We suggest other possible uses, including traffic accounting, traffic routing, and in general any other packet classification process based on connections, and which can take advantage of the architecture's simple, generic mechanisms to classify traffic.

4.2 Introduction

This Chapter describes Shunting. Shunting is a novel architecture to perform high-speed, extensive (non-sampled and in-depth), stateful, inline traffic processing. We will frame Shunting in the context of carrying out Network Intrusion Prevention in high-speed links (1 Gbps and above).

Network Intrusion Detection System (*NIDS*) and Network Intrusion Prevention System (*NIPS*) are systems that detect malicious network activity (denial of service attacks, port-scans, etc.) by monitoring network traffic [Mukherjee et al., 1994]. The main difference between NIDS and NIPS is their reaction to the detection of

such malicious activity. While NIDS are passive systems that fire alerts and log attacks, NIPS are reactive systems: They respond by blocking attacks, dropping packets deemed hostile while letting innocuous packets go through.

NIPS and NIDS are typically used to defend an organization network from external attacks. This means they are typically located in the link that connects such an organization to the Internet.[1] This also means they do not provide any defense against attacks against local hosts initiated by other local hosts.

Inline, rich per-packet monitoring of a high-speed link to detect security intrusions is a resource-intensive task. Each packet must be captured, analyzed, and forwarded if deemed innocuous. The analysis part, i.e., deciding whether a packet poses a security threat or not, may require a considerably complex effort. Operational use in a high-volume environment intensifies the problem by introducing performance-related complications.

The straightforward system troubles caused in NIDS by an increase in the amount of traffic are magnified by two other effects, namely traffic diversity and state explosion. First, as the amount of traffic increases, the traffic diversity and the crud in the link also increase, which produces not only more false alarms, but also more diverse ones [Dreger et al., 2004].

Second, NIDS that want to understand the traffic they are seeing need use state. When receiving a new packet, a NIDS must consider it in the context of existing

[1]This zone is typically known as the *DMZ*, from "DeMilitarized Zone," a military term that describes a buffer area between two enemies.

information on the packet's connection. This context is obtained from already-received packets from the same connection, which the NIDS must have stored. Connection-oriented dependencies not only extend to the past, but also to the future: The NIDS may not be able to complete the processing of a packet until it receives further traffic from the connection. This statefulness requirement gets exacerbated by the existence of attacks based on stack or topology ambiguities [Ptacek and Newsham, 1998].

In stateful NIDS, the amount of state needed to produce a good snapshot of the network state is proportional to the amount of traffic. This creates an enormous state explosion problem.

4.2.1 Shunting in a Nutshell

To address this problem, we introduce Shunting. Shunting is a new packet processing architecture that provides a software analyzer with a tool to process traffic at very high speeds. The processing Shunting offers is based in tables, which while simple have three main advantages: They can be implemented in fast hardware, they can be programmed dynamically, and they provide the right granularity the analyzer needs to process traffic (namely connections, addresses, and ports).

The basic idea of Shunting is to have a simple-but-fast hardware device that performs table lookups as the front-end of a complex-but-slow traffic analyzer. Traffic is processed first by the hardware device, which looks up the packet in its tables, and takes a very simple decision, namely whether the packet has to be forwarded directly,

dropped, or diverted to the analyzer (*shunt* the packet).

The analyzer receives the shunted traffic, processes it, and maybe reinjects it back to the device. The analyzer also updates the contents of the device tables, therefore driving the device policy.

In order for Shunting to be effective, the processing must be such that most of the traffic ends up being processed only by the hardware device (forwarded or dropped, but not shunted) . This implies that the analyzer physically receives a much smaller stream than the one it is processing.

By making the hardware-device decision mechanism as simple as a table lookup with a handful of possible yields, we are trading off simplicity in exchange of temporal and spatial efficiency, in other words, being able to do very fast lookups, and providing very large tables, fitting millions of entries.

Shunting permits extensive, stateful, inline traffic processing in several packet processing scenarios, including intrusion detection. The processing that Shunting permits has three properties: First, it is extensive. Shunting does not resort to sampling (unless specifically requested by the user), and permits in-depth, rich per-packet analysis. Second, it is stateful. The analyzer may store state from the processed packets. Last, it is inline. Therefore, it allows performing intrusion prevention ("bump in the wire" processing).

Shunting is carried out using commodity PC hardware, and a simple, special-purpose hardware device.

Shunting is based on a simple observation: Packet processing engines may have to perform an intensive analysis to decide how to process a single packet, but they are often able to dynamically decide that a subset of all future traffic can be processed with minimal, simple analysis. This analysis is automated into a simple, hardware device, where the engine can offload work out of itself. This produces the effect of limiting the resource consumption of the engine without reducing the amount of traffic processed.

An example of the aforementioned observation can be drawn from the intrusion detection world. Let's assume a NIDS is monitoring the application-layer payloads of an SSH connection.

The NIDS is the decision engine in the Shunting model. It processes packets, and decides what to do with them. For the NIDS, however, application-layer contents are only useful until the connection gets encrypted. If, at that point, the NIDS is able to label the connection as malicious, the processing of all the remaining connection traffic is as easy as "drop any packet corresponding to this connection." Otherwise, as any further traffic is encrypted, analyzing it is useless for intrusion detection purposes. The best way to process all the remaining connection packets is "forward any packet corresponding to this connection."

The rest of the chapter is organized as follows: Section 4.3 introduces the intrusion detection problem through a related-work discussion. Section 4.4 presents Shunting in an in-depth fashion, and justifies its rationale. It also discusses several applications,

including intrusion detection. Section 4.5 describes the Shunting design, including a preliminary implementation for carrying out intrusion detection on a high-speed link. Sections 4.6 presents an evaluation of the Shunting architecture. Sections 4.7 discusses some future work, and Sections 4.8 concludes.

4.3 Related Work

This Section is structured as follows: Section 4.3.1 introduces Intrusion Detection Systems, both Host-Based and Network-Based. Shunting is an architecture oriented to the second type, for which Section 4.3.2 provides a taxonomy, and Section 4.3.3 a description of some examples, including Bro, where we have implemented Shunting.

Section 4.3.4 describes a fundamental problem in NIDS, namely the existence of ambiguities. A consequence of the mechanisms used to deal with ambiguities, especially in high-speed networks, is the need to manage resource exhaustion, described in Section 4.3.5. Shunting is a technique to help manage resource exhaustion in NIDS.

Section 4.3.6 describes research in NIDS parallelization, which has a strong influence in our work. Section 4.3.7 describe the use of hardware support for fast packet processing, from which Shunting draws heavily. Section 4.3.8 describe the fine-tuning of the software side of network adapters, which is complementary to the previously mentioned research.

Section 4.3.9 compares the Shunting filtering model with those of traditional packet filtering models.

Section 4.3.10 discusses research in heavy-tailed evidences in network traffic, and how it justifies our choice of the shunting actions.

4.3.1 Intrusion Detection Systems

The goal of Intrusion Detection Systems (*IDS*) is to detect attacks on computers.

There are two main types of IDS, namely Host-based Intrusion Detection Systems and Network-based Intrusion Detection Systems.

Host-based Intrusion Detection Systems

The first well-known approach for detecting attackers is Host-based Intrusion Detection Systems (*HIDS*) [Denning, 1987]. HIDS run on the end-hosts they are protecting, monitoring such activities as session logins [Denning, 1987], system calls [Bernaschi et al., 2000; Forrest et al., 1996], program execution [Denning, 1987], file access [Denning, 1987; Pennington et al., 2003], etc., searching for anomalous behavior. A user, program, or system behavior is considered "anomalous" when it differs substantially from a "normal" behavior model. The latter can be generated manually [Bernaschi et al., 2000] or automatically [Denning, 1987; Forrest et al., 1996], including through static analysis of the program source [Wagner and Dean, 2001].

In a seminal work proposing the idea of HIDS, Denning described IDEA, a model of a real-time intrusion detection expert systems [Denning, 1987]. Denning's idea consists of a set of tools that first creates a statistical model of the users' behavior

(session logins, program execution, and file access), and then tries to detect anomalies in the actual behavior. Such anomalies are considered a signal of computer abuse.

The HIDS idea is related to the notion of computer introspection, in which a system or a program develops a model of what it should be doing and/or what it should not, and tries to identify the latter.

Network-based Intrusion Detection Systems

Network Intrusion Detection Systems (*NIDS*), on the other hand, try to detect attacks on computers by monitoring the network [Mukherjee et al., 1994]. A NIDS typically operates by observing the traffic packets or connections as they flow through the network, trying to detect malicious network activity, such as service attacks or port scans.

The best advantage of HIDS, as compared to NIDS, is that their location in the monitored host makes them resilient to evasion techniques based on ambiguities at the network, transport, and especially application protocol [Paxson, 1999; Ptacek and Newsham, 1998]. In that sense, HIDS enjoy better and broader visibility of the attack than NIDS: They can see the attack at different stack levels, and therefore resolve the ambiguities the same way the host does. Finally, their workload is much smaller than that of NIDS.

HIDS present two main cons. First, they run on the same host they are trying to defend, so the defender (HIDS) is not independent of the defended one (the host). This

implies that HIDS are, at most, as resistant as the host they defend. Crash or subvert the host, and the HIDS will become completely useless. To combat this problem, some researchers propose running HIDS as a Virtual Machine Monitor (*VMM*), so that the HIDS will be isolated from an attack on the monitored host [Garfinkel and Rosenblum, 2003].

Second, you must install one HIDS for every host you want to defend, which not only is cumbersome, but also may include porting the NIDS to a wide variety of hosts. In comparison, a single NIDS can be used to monitor networks composed of several thousand, heterogeneous, diversely-administered hosts.

4.3.2 NIDS Types

There are three principal types of NIDS: anomaly-based, specification-based, and signature-based NIDS.

- Anomaly-based NIDS (*A-NIDS*) look for unusual behavior in the network activity [Gil and Poletto, 2001; Jelena and Greg, 2002]. They use a database of normal behavior profiles, usually adapted to the network they are protecting, plus a set of statistical methods to detect unusual behavior in new traffic. Such unusual behavior is considered a signal of maliciousness. A-NIDS typically learn what normal behavior is in an automatic fashion, by being trained with normal network activity. Inferring automatically what is normal behavior is a fundamental feature of A-NIDS.

- Specification-based NIDS (*Spec-NIDS*) are provided with an specification of what is legal behavior, and therefore allowed [Ko et al., 1997]. In some sense, Spec-NIDS are manual A-NIDS. What is allowed or not is not inferred from seeing normal traffic, but directly specified by the user. This makes Spec-NIDS more reliable than A-NIDS, at the cost of being more labor-intensive. (The specifications must be encoded.) The main advantage of SpecNIDS is that they are able to detect zero-day attacks.

- Signature-based NIDS (*S-NIDS*), on the other hand, look for known patterns of attacks (known as *signatures*) inside the traffic they are monitoring. They are also known as misuse detectors, pattern detectors, or packet *greppers*. S-NIDS use a database of attack signatures, expressed as connection or packet contents. Traffic is compared to the database, and if any matches, it is considered a signal of maliciousness.

The three NIDS flavors must deal with a tradeoff concerning the tightness of their attack definitions (normal behavior profiles in A-NIDS and Spec-NIDS, attack signatures in S-NIDS). Tighter definitions risk missing slight variations of a well-known attack (false negatives), or misinterpreting slight variations of good-behaved traffic (false positives). Looser definitions risk matching perfectly valid or malicious traffic, therefore increasing the amount of false positives or false negatives.

S-NIDS are typically more precise than Spec-NIDS and A-NIDS. The reason is that S-NIDS work with signatures, i.e., explicit information of how an attack looks

like. Assuming the signatures are distinctive enough, a match implies strong evidence of an attack. Spec-NIDS and A-NIDS, on the other hand, use deviations of normal behavior as a proof of malice. This means they may produce false positives when the behavior of some perfectly valid traffic is unusual enough.

S-NIDS are more limited in scope and more static than the other two. S-NIDS have no means to detect novel attacks, or variations of previously-known attacks, as they lack a signature for the attacks. Spec-NIDS and A-NIDS, on the other hand, may detect some previously unknown attacks, provided the attack behavior is unusual enough.

S-NIDS are more cooperation-friendly. S-NIDS signatures can be described in a simple form, which makes it easy to share them among different NIDS.

The last difference between the three types of NIDS is the type of information decisions are based on. While S-NIDS' tight definition of attacks typically limits them to packet-content or connection-content grepping, A-NIDS and Spec-NIDS' adaptable definition of normality permits them to use more information sources than just contents, such as inter-packet timing and size [Zhang and Paxson, 2000a], address correlation [Zhang and Paxson, 2000b], traffic meaningfulness [Staniford et al., 2002a], host "promiscuity", flow rates [Jelena and Greg, 2002], flow rate asymmetry [Gil and Poletto, 2001], etc.

4.3.3 NIDS Examples

Two well-known open-source (and therefore suitable for study) NIDS are Snort [Roesch, 1999] and Bro [Paxson, 1999].

Snort [Roesch, 1999] is a popular, open-source, Signature-based NIDS. Snort is basically *tcpdump* with pattern matching. It captures a subset of the traffic using *libpcap*, and then compares it to a set of pattern-matching rules. When any of the patterns is matched, Snort raises an alert or logs an event, depending on the rule definition.

Snort permits specifying unused hosts and ports, and will report activity on any of them. Snort also supports IP fragmentation reassembly, via the *frag2* preprocessor, and TCP segment reassembly.

Finally, Snort supports state timeout policies to avoid attacks based on state accumulation.

Bro [Paxson, 1999] is another well-known open-source NIDS. It is of special importance for this thesis, as we are using it as the network analyzer that drives the Shunting system.

Bro is a mixture of Anomaly-based, Specification-based, and Signature-based NIDS. Bro generates events the reflect the activity in the network, which are used by intrusion detection-oriented analyzers. In some sense, Bro is a mechanism where different policies (analysis types) can be performed.

Bro's basic model consists of three layers: packet filtering, event engine, and policy

script interpreter.

The first step is packet filtering: Bro uses libpcap to specify which traffic it knows and wants to analyze, and therefore separate it from the remaining traffic.

The second step is the event engine. The event engine performs analysis of traffic at network-, transport-, and analysis-layer protocol, including IP defragmentation, and checking that the packets are well-formed. Upon analyzing the traffic, the event engine generates a set of events, established at different semantic levels.

Table 4.1 shows a list of selected Bro events.

layer	event name	raise for every ···
network	new_packet	new IP packet
transport	connection_established	new full TCP handshake
transport	connection_attempt	TCP connection where the origin SYN has not been followed by a SYN/ACK
transport	connection_timeout	TCP connection for which some required activity has not been seen
transport	udp_request	UDP packet whose port has no application-layer analyzer associated
application	http_request	new HTTP request
application	http_reply	new HTTP response
application	ftp_request	new FTP command
application	ftp_reply	new FTP reply

Table 4.1: Selected Bro Events

The idea of the event engine is to provide a NIDS framework, i.e., a generic mechanism where NIDS policies can be implemented. In that sense, the event engine is policy neutral, and it cannot be qualified as neither Signature-based, Specification-based, or Anomaly-based NIDS. It supports the three of them. Of course, the definition of the events (the mechanism itself) introduces a great deal of shaping in the policy

possibilities.

The last step is the policy script interpreter. Bro defines a NIDS-oriented language that permits users to specify policies on what the NIDS should do to respond to an event. The policy script writer can make use of rich data types, persistent state, timers, and external applications. This way she can incorporate as much context as she needs in order to decide how to react to the event, which can include updating the state or generating alerts.

4.3.4 Ambiguities and Evasion Techniques

A fundamental problem for passive NIDS is the existence of ambiguities in the traffic stream [Handley et al., 2001], which make unclear how to interpret it. Ambiguities originate because of three different causes [Handley et al., 2001; Ptacek and Newsham, 1998]:

- Incomplete NIDS: A NIDS must be able to analyze the complete range of options for every protocol.

  An example of this type of ambiguities is dealing with IP fragmentation. This is a cumbersome process, as it requires storing and reassembling fragments in the NIDS. As a consequence, some NIDS do not correctly reassemble fragments [Ptacek and Newsham, 1998].

  An example of network-layer, incomplete-NIDS ambiguity is dealing with IP fragmentation. Some NIDS are unable to reassemble IP fragments, or to reorder

out-of-order IP fragments. Fragmented or out-of-order fragments are therefore processed incorrectly by the NIDS.

An example of transport-layer, incomplete-NIDS ambiguity is dealing with TCP segment reordering: Some NIDS are unable to reorder out-of-order TCP segments. Out-of-order TCP segments are processed incorrectly by this type of NIDS.

- Stack-based ambiguities: Some protocols specifications do not specify their behavior exhaustively. As a consequence, different end hosts behave differently. This get complicated by wrong or incomplete implementations of such protocols.

  The network and transport protocols are the most common targets, although application protocols (Layer 7) can also be attacked.

  An example of network-layer, stack-based ambiguity is inconsistencies in IP fragments. The IP specification [Postel, 1981a] does not state what the receiving side should do when confronted with two overlapping fragments whose contents are inconsistent. Some stacks use the "first byte ever" principle, some use the "last byte ever" principle, and others use a mix. An example of ambiguity relates to the use of uncommon TCP-flag combinations. When confronted with a segment with some specific TCP-flag combinations, some stack implementations drop the segment, while others accept it.

  Some examples of transport-layer, stack-based ambiguities include: (a) TCP

segmentation inconsistency: The TCP RFC [Postel, 1981b] does not state what the receiving side should do when confronted with two overlapping segments whose contents are inconsistent. Some stacks use the "first byte ever" principle, some use the "last byte ever" principle, and others use a mix. (b) TCP options: Another source of ambiguities is the use of TCP options, which are not mandated by the TCP specification [Postel, 1981b]. A case example is the use of *PAWS* (Protection Against Wrapped Sequence Numbers) [Jacobson et al., 1992]. The PAWS TCP option is a mechanism to reject old duplicate segments that might corrupt an open TCP connection. If the victim interprets PAWS and the NIDS does not (or vice versa), a carefully crafted segment will be rejected by the victim, while accepted by the NIDS.

An example of L7 stack-based techniques is HTTP Request Smuggling (HRS), where different and/or wrong implementations of HTTP persistent connections are used to cause different HTTP traffic views at the end-host and diverse middleboxes (web cache, web proxy, firewall, etc.) [Linhart et al., 2005]. HRS can be used to poison web caches, evade IDSs, and, when combined with a script vulnerability in the server, request hijacking at a proxy server.

- Topology-based ambiguities: In some cases, it is the fundamental operation of an efficient networking protocol which causes the ambiguities.

  An example is TTL-based ambiguities. The TTL field in an IP packet states

how many more hops the packet can be forwarded before being discarded. The objective of this field is to avoid that routing inconsistencies or misconfigurations forward packets around in the network indefinitely. Routers decrement by one the TTL field of any packet they forward. If the TTL field reaches zero, the packet is dropped, instead of forwarded.

TTL-based ambiguities occur when the NIDS sees a packet whose TTL is small enough that it may or may not reach the end host.

Other topology-based ambiguities relate to (a) path MTU: If the path MTU narrows between the NIDS and the end host, packets sent with the "Don't Fragment" bit set to 1 may not reach the end host (it depends on whether the packet size is big enough that it does not fit in the narrower path). (b) bandwidth: If there is a slow and congested link between the NIDS and the victim, low-priority packets will be dropped in the path between the NIDS and the end host.

Note that, while incomplete NIDS ambiguities can be solved by creating better NIDS, stack-based and topology-based ambiguities are fundamental, and NIDS can not get away from them by using pure passive analysis.

Evasion Techniques

A problem faced by NIDS is attacks that explicitly target the NIDS. These attacks can be as harsh as overloading or even crashing the NIDS (e.g., Denial of Service

attacks), and as subtle as Evasion [Ptacek and Newsham, 1998]. Evasion is based on taking advantage of ambiguities. It consists of an attacker forging data traffic with the explicit purpose of duping and/or attacking the NIDS to evade its detection. Evasion is today a reality [Song, 2001].

Evasion techniques take advantage of the fact that ambiguities in communication protocols may cause the NIDS and a victim end-host to react to the same traffic in different ways.

Evasion techniques come in two flavors, insertion and evasion. An Insertion attack consists of fooling the NIDS into accepting data the victim end-host will reject. Evasion is based on the opposite idea: the NIDS is fooled into rejecting data that the victim end-host will accept. The objective in both cases is the same: The attacker will create a view on the connection data at the NIDS different from that at the victim.

Note that both techniques can often be used to create the other. For example, if an attacker can insert a packet into the NIDS that the victim host will not see, she can use it to insert a forged RST segment. The NIDS will think that the connection has been torn down. For the NIDS, any further segment from the same connection is spurious, and therefore will be not considered. In the victim's eyes, further segments are perfectly valid. Therefore, further connection packets are evaded from the NIDS, and inserted into the victim.

The same approach may often be used to turn evasion into insertion. By evading

an RST segment from the NIDS, an attacker will make the victim end-host declare the connection closed. The victim will drop any further packet from that connection, which the NIDS will accept.

Interestingly enough, the techniques that permit evasion techniques can be used for fingerprinting. Stack-based techniques are routinely used to fingerprint hosts (*nmap*), Topology-based techniques are used to fingerprint networks (*traceroute*).

Fighting Ambiguities

Assuming that it implements all required protocol specifications, there are several approaches to combat ambiguities:

- Normalization [Handley et al., 2001] consists of introducing an inline network element to patch up (normalize) the packet stream in order to remove potential ambiguities. As discussed by [Handley et al., 2001], there are several issues with normalization.

  First, normalizing traffic modifies the end-to-end semantics of a connection. While some modifications are probably harmless (dropping inconsistent, overlapping fragments), some affect perfectly legal, useful traffic (low TTLs are the basis of traceroute).

  Second, normalizing traffic can affect the end-to-end performance of a connection. For example, removing the TCP window scale option decreases the performance of a TCP transference.

Third, the view from a normalizer can be fundamentally incomplete to detect an attack. For example, the semantics of TCP urgent pointers depend on the application. If an attacker sends a segment with the text "robot," with the URG pointer pointing to the 'b' character, the receiving end application will receive "robot" or "root" depending on which options were used to open the socket.

Last, normalizing traffic requires keeping state to detect some of the ambiguities. This means an attacker may try to crash the normalizer by instantiating lots of state. An example is to send multiple, inconsistent IP fragments, but never completing a full packet.

Shunting can act as a generic inline packet processor, and therefore serve as a framework for normalization, if the NIDS driving the Shunting architecture supports it.

- Active-Mapping [Shankar and Paxson, 2003] consists of gathering information on the hosts comprising the intranet being monitored, and using this information to solve ambiguities. Active mapping assumes that the monitored intranet is relatively stable. It generates a profile of the way all internal hosts solve stack ambiguities, and the internal network topology (hop counts and path bandwidths).

  When receiving an ambiguous packet, the active mapper checks the profile of the target host, and then decides the exact meaning of the packet. Shankar

and Paxson report being able to map a single host in 35 seconds with 19 KB of traffic, using just 100 bytes of space per host.

- Exhaustive Analysis, also known as bifurcating analysis, consists of storing all the ambiguous data, and analyzing the traffic following all the reasonable combinations of the different ambiguity resolutions. This presents an important processing problem, as the analysis may grow exponentially with each new ambiguity. It also presents a storage problem, as the state may grow proportionally to the amount of data in every connection.

4.3.5 Resource Exhaustion Management

One of the main problems with sophisticated NIDS is resource exhaustion [Dreger et al., 2004]. The problem is two-fold: for stateless NIDS, the main problem is CPU load. For stateful NIDS, the problem is state explosion.

This problem is made worse when operating NIDS in high-speed environments. Dreger *et al.* provide some operational experience when running Bro in Gigabit networks [Dreger et al., 2004].

Two common solutions to run NIDS on high-speed environments are state management and input-volume control techniques [Dreger et al., 2004]. They are complementary: Input-volume control techniques limit the amount of traffic that the NIDS must process, while state management techniques manage the state created by the processed traffic.

State Management

State-management techniques are intended to limit the amount of state kept by the NIDS [Dreger et al., 2004; Paxson, 1999]. This includes using timeouts, fixed-size buffers, compressing state, and checkpointing.

The goal of timeouts is to perform implicit state removal. Some transport protocols, such as TCP, signal the end of a connection explicitly in the wire. Therefore, the NIDS can take advantage of a TCP connection FIN/RST handshake to safely remove all the information related to that connection. In other words, the NIDS can explicitly erase the state associated to the connection. Other transport protocols, as UDP, do not signal the end of a session in the wire. Therefore, there is no generic way for the NIDS to realize an UDP connection is terminated, except for the absence of new datagrams. The idea of timeouts is to delete state corresponding to connections that have not shown activity for a while. This also covers TCP connections where the RST/FIN segments have been lost.

The rationale behind fixed-size buffers is similar to that of timeouts: Old state is less valuable than new state, and therefore the NIDS may dispose of it easily. The difference is that fixed-size buffers only evict state when needed. In the fixed-size buffers technique, the NIDS allocates a fixed chunk of storage for all the state. Removal of state occurs only when a new piece of state must be created, but the total storage chunk is full. In this case, an old piece of state is selected for removal. (FIFO is a sensible approach.) The main advantage of fixed-size buffers is that there

is a guaranteed hard limit in the amount of state the NIDS requires, namely the storage chunk size. The main disadvantage is that fixed-size buffers evict state when they need to, not when the state it old or unused (and therefore probably less useful). This means underestimating the amount of allocated storage may cause thrashing, while overestimating it may limit the benefit of the state limitation technique (the NIDS may end up keeping very old information, as there is no need to evict it).

Compressing state takes a similar approach, focusing on avoiding state creation whenever possible. For example, Bro only creates full per-connection state when it sees activity (a packet) from both endpoints of a connection. This way, it avoids creating a connection state entry for every unanswered connection attempt, which diminishes the probability of being overwhelmed by state during flooding attacks, large worm events, or simple portscans.

Related to state management, [Sommer and Paxson, 2005] proposes the use of independent state in NIDS. The goal of independent state is to provide the ability of extracting the NIDS state out of the process it exists into. This permits the state being shared between concurrent instances of NIDS (spatially independent state), and continuing to exist after the process terminates (temporally independent state) [Sommer and Paxson, 2005]. Independent state can also be used to help parallelize NIDS processing among several NIDS instances (see Section 4.3.6).

A similar piece of work in state management is checkpointing [Paxson, 1999; Sommer and Paxson, 2005]. NIDS stability tends to diminish with time, as more

state is kept. The idea of checkpointing is basically restarting the NIDS periodically, resetting the state, and giving the NIDS a fresh start. It is a very coarse-grained approach.

Input-Volume Control Techniques

Input-Volume control techniques work by reducing the amount of traffic the NIDS processes, limiting the analysis to only part of the traffic. This limitation can be static or dynamic.

The static limitation is quite simple: The NIDS configures the packet filter so that it gets fed only with a subset of the traffic. This subset may include what the NIDS is interested in, or what it actually can process. For example, a NIDS should only request receiving traffic from those ports whose standard services it knows how to process.

Another static approach consists of sampling the input stream, either in a packet- or a connection- basis. (Some analysis require accessing to full connections.)

A more powerful tool to reduce state is being able to set the packet filter dynamically. Two mechanisms that dynamically limit the NIDS input volume are load-levels [Dreger et al., 2004; Lee et al., 2002] and flood detectors [Dreger et al., 2004].

The idea of load-levels is to extend the NIDS with a set of ordered packet filters. Each filter is more restrictive that the previous one, therefore providing a smaller input volume. The NIDS senses its workload, switching to a more restrictive filter

when it feels overwhelmed, and to a less restrictive filter when it feels idle.

A flood detector tries just to shun from the NIDS all the traffic related to Denial of Service floods directed to a single host. This technique consists of detecting the flooders, and shunning them from the NIDS. It leverages the fact that flooding has no other meaning than overwhelming a resource, and therefore further analysis is not required.

Shunting is an example of dynamic input-volume control technique. Shunting permits NIDS to specify the traffic they want to analyze in a very fine grained way. It does so using per-connection, address, and port tables. It is a more dynamic and fine-grained approach than load levels. The address table permits the efficient implementation of the flood detector.

The resource exhaustion problem gets exacerbated in inline packet processors (as Shunting), as packet drops are a harder problem here: If an (offline) NIDS drops packets, it will only have a blurrier/more limited vision of the traffic. Assuming that the amount of malicious traffic is a small fraction of the whole traffic, and that the attack traffic is not correlated with the drops[2](i.e., that the probability that a packet is dropped does not depend on whether it is malicious or not), the probability that it misses an attack is low, and even if it happens, it merely causes a false negative.

A NIPS, by definition, is an inline element. If a NIPS drops packets, it will result in a packet loss in a connection. This will cause throughput loss, not only because of

[2]Of special importance in this case is that the attacker herself is not able to correlate her traffic with the drops.

the consequent retransmissions, but also, and especially, because of the interference with transport-protocol congestion-avoidance mechanisms.

4.3.6 NIDS Parallelization

Yet another idea to avoid resource exhaustion is to implement the NIDS as several hosts running in parallel, and divide the work among them.

Kruegel *et al.* implement a high-performance, signature-based NIDS by collocating several NIDS in parallel [Kruegel et al., 2002]. The authors propose a 4-step process that manages to provide each NIDS with a subset of the total traffic that conforms to a small superset of the traffic it needs to detect an attack.

1. Input traffic is plugged first into a fast, simple piece of hardware ("scatterer"), which divides the traffic evenly (round-robin) among a group of classifiers ("slicers").

2. Every slicer has a complete list of all the signatures supported by the system. Slicers check every packet, identify suspicious ones, and forward them to the corresponding reassembler. Note that, if a packet matches more than one signature, it may end up being forwarded to several reassemblers.

3. Reassemblers fix the packet stream (they may receive out-of-order packets, if two packets are dispatched by different slicers) before it is passed to the different NIDS engines.

4. Sensors are the NIDS engines. They are assigned a portion of the signatures,

and they receive a subset of the total traffic, on which they run the signature matching.

Shunting can be used to parallelize the intrusion detection work among several instances of NIDS. The granularity offered to divide tasks is connections, addresses, and ports. In comparison, the granularity offered by [Kruegel et al., 2002] is string matching. As a consequence, Shunting is not limited only to Signature-Based NIDS, and can be also used in Anomaly- and Specification-Based NIDS.

4.3.7 Hardware Support for Packet Processing

In order to process traffic at high-speed links, a traditional solution is to use some hardware support. This is commonly known as "pushing processing to the network adapter."

Shunting also uses hardware support, which we term the "hardware device." Our hardware device is used to perform fast table-based filtering. A similar idea has been suggested to perform passive packet capture (see Section 4.3.7) and to offload transport-protocol work (see Section 4.3.7).

Pushing Processing to the Network Performance Card

There are several network adapters that perform complex packet processing, therefore permitting the host in which they are located to achieve high-speed packet processing by pushing processing to the adapter.

Juniper routers permit filtering packets based on (at least) source and destination IP address and port, packet type, protocol, length, ICMP type and code, VLAN ID, TCP flags, and fragments [Markatos, 2005].

Deri proposes to perform high-speed passive packet monitoring using a router (Juniper M-series, which allows for traffic filtering based on header fields) that acts as a smart Network Interface Card (*NIC*), performing generic traffic accounting and simple packet filtering and sampling, and sending the filtered/sampled stream to a Linux host [Deri, 2003].

The Intel IXP family of "network processors" provides a framework to perform in-NIC packet-processing [Intel, 2005]. The IXP1200 series is composed of six RISC processors (aka microengines) that operate in parallel, and a StrongARM control processor running Linux. Each microengine has 1 KB of instruction storage, some registers, and four contexts. There is also a shared 4 KB scratch space [Markatos, 2005].

Endace's DAG cards are PCI-based network adapters specialized in passive monitoring of high-speed links [Cleary et al., 2000]. While they are able to capture full packets, the common approach used to make DAG cards capture high-speed traffic is to instruct them to capture only the first few bytes (the network- and transport-layer headers) of every packet [Markatos, 2005]. This approach is commonly known as "pushing the snaplen into the network adapter," and while it is fine in some scenarios, it is unacceptable in NIDS.

A DAG card consists of (a) a programmable FPGA, which generates high-precision timestamps (possibly coming from a GPS device), parallelizes physical layer bytes into 32-bits words, filters out unwanted data, buffers packets in a FIFO before sending them to the host, and counts dropped packets; (b) an ARM based CPU (ARM7 in DAG 2 cards, StrongARM in DAG 3 ones); and (c) a PCI interface to communicate with the host PC.

Using Endace's DAG 4 cards, Iannaconne *et al.* show a network adapter that may permits passive monitoring of OC-192 links (10 Gbps) [Iannaccone et al., 2001]. The authors' idea is to use a specialized piece of hardware (the DAG card's on-board FPGA) to compress the snaplen-reduced packets into flow traces, and only send those flow traces to the PC host. The authors use a hashed, limited-size connection table to store the flow traces, and claim what, with the help of fast PCI buses (64 bits, 66 MHz), it is possible to monitor IP, TCP, and UDP headers (throwing the rest of the packets) in 10 Gbps links.

The SCAMPI project proposes using a smart network adapter to limit the amount of traffic that reaches the host in packet capture scenarios [Coppens et al., 2003, 2004]. SCAMPI runs on several different architectures, including Intel IXP family of network processors, Endace's DAG cards, and their own network adapter, called "COMBO." COMBO adapters perform systematic (deterministic) and probabilistic 1-in-N sampling, address- and port-based sampling, payload string searching, generic flow-state accounting and reporting, and packet filtering using FPL-2 (an extended, BPF-like language).

In comparison with all the mentioned approaches, Shunting is much simpler, only permitting filtering based on the three aforementioned tables. The main benefits of simplicity are space and efficiency. Simple lookup-based processing permits very-large tables, which match the requirements of large-scale connection-based processing, and easy parallelization of the processing, which causes an increase in packet processing throughput.

NIC Offloading Processing

The idea of offloading work to the network adapter has also been proposed in the context of network protocol implementation, a field related to packet processing. The idea is also known as "Protocols in Silicon [Clark et al., 1989]," and consists of instrumenting the interface card to do part of the network or transport protocol stack processing for the CPU. The whole set of techniques to offload network stack work to the NIC are also known as "TCP Offload Engines [Currid, 2004; Mogul, 2003]," or TOE.

Protocol stack offloading has been proposed in several forms. One is "interrupt coalescing," where the interface card is instrumented to wait some amount of time after a packet arrives, with the hope that the card can serve many packets with just one interrupt.

Another TOE technique, and probably the most popular, is "Checksum Offloading," in which the interface card checks the network and transport checksums of incoming

packets, and calculates the network and transport checksums of outgoing packets [Kleinpaste et al., 1995].

Other TOE techniques include moving to the NIC the updating of stack state (TCP sequence and acknowledge numbers), timers, segmenting and reassembling, buffer management, scattered copies, etc.

4.3.8 Software Support for Packet Processing

Some researchers have focused on optimizing the software side of the packet processing tools.

The traditional approach to packet capture is the use of interrupt-driven systems. In such systems, when a packet arrives, the NIC interrupts the CPU. The corresponding interrupt service routing (*ISR*), which we will name the "hardware ISR," does some initial packet processing, places the packet on a queue, and finally generates a software interrupt. Some time later, the corresponding ISR (the "software ISR") is dispatched. It processes the packet in full, including filtering, if captured through a packet filter.

When running at high speeds, interrupt-based packet capture systems may suffer "receive livelock" [Mogul and Ramakrishnan, 1997]. For historical reasons[3] the software interrupt has less priority than the hardware one. In high-speed scenarios, the receiver services the hardware ISR, and before it manages to service the software one, another packet interrupts it. The backlog of software ISR keeps increasing, until the driver

[3]Old NICs had little buffer memory, and therefore it was crucial to move packets as fast as possible out of the NIC and into memory.

buffers get full. When the hardware ISR finds the driver buffers full, it just drops the packet. Therefore, the receiver spends all its time processing interrupts, and no packets are ever delivered to the user application. The resulting throughput is zero.

[Mogul and Ramakrishnan, 1997] proposes some solutions to avoid receive livelock: (a) interrupt coalescing: It consists on batching several interrupts, so that several packets are processed with the help of just one interrupt; (b) limiting the interrupt rate: If the system detects too many hardware interrupts, it disables them temporarily; and (c) avoiding preemption of the software ISR by the hardware ISR.

A modern approach to packet capture is the use of device polling [Mogul and Ramakrishnan, 1997; Morris et al., 1999]. In such a system, the kernel polls the device's receive DMA queue periodically, in case there is a newly arrived packet. The traditional argument against polling is that it causes a large overhead when no traffic is received. [Mogul and Ramakrishnan, 1997] proposes to use a combination of both, so interrupts are used during low loads, and polling during high loads. [Morris et al., 1999], on the other hand, proposes a pure polling approach, arguing that "even infrequent PC interrupts are simply too expensive" in modern PCs.

Another related approach to enhance the performance of software packet-capture systems is to minimize the number of copies of packet buffers. A suggested approach is to share buffers between the application finally processing the packets and the kernel [Wood, 2004]. A bolder approach is to give applications full control over the NIC, including the card registers, so the applications can do the polling themselves

[Cleary et al., 2000]. This introduces several problems related to synchronization when there are multiple readers [Degioanni and Varenni, 2004].

Optimizing the software side of packet processing tools is complementary to Shunting. The goal of Shunting is to be narrow the stream that reaches the analyzer, and therefore the amount of job that the NIC must perform. Still, the narrowed stream may be big enough as to require software tuning in the packet capture processing.

4.3.9 Filtering Models

Another related piece of work is research in filtering models. Section 3.3 in Chapter 3 describes previous work in packet filtering, focusing on BPF, the most common packet filter.

Shunting provides a much simpler filtering model than traditional packet filters. Instead of providing a generic, virtual processor with most of the operations available in normal processors, shunting offers a very simple, stateful, dynamically-programmable table lookup mechanism based on three simple tables.

Shunting renounces to the flexibility existent in traditional packet filtering models, in exchange of being able to leverage very fast, parallel table lookups in a hardware device.

4.3.10 Network Traffic Heavy-Tailed Evidence

The last piece of related research is the multiple evidences of the self-similarity in network traffic [Crovella, 2001]. In particular, connection sizes have shown to follow a heavy-tailed (power-law) distribution [Crovella and Bestavros, 1996; Paxson, 1994; Paxson and Floyd, 1995], with the heavy-tailed distribution of data object sizes being suggested as the underlying cause of it [Crovella and Bestavros, 1996].[4]

Figure 4.1 shows the bytes in the tcp-1 trace as a function of fraction of smallest connections. Less to 0.4% of the connections account for more than 90% of the bytes.

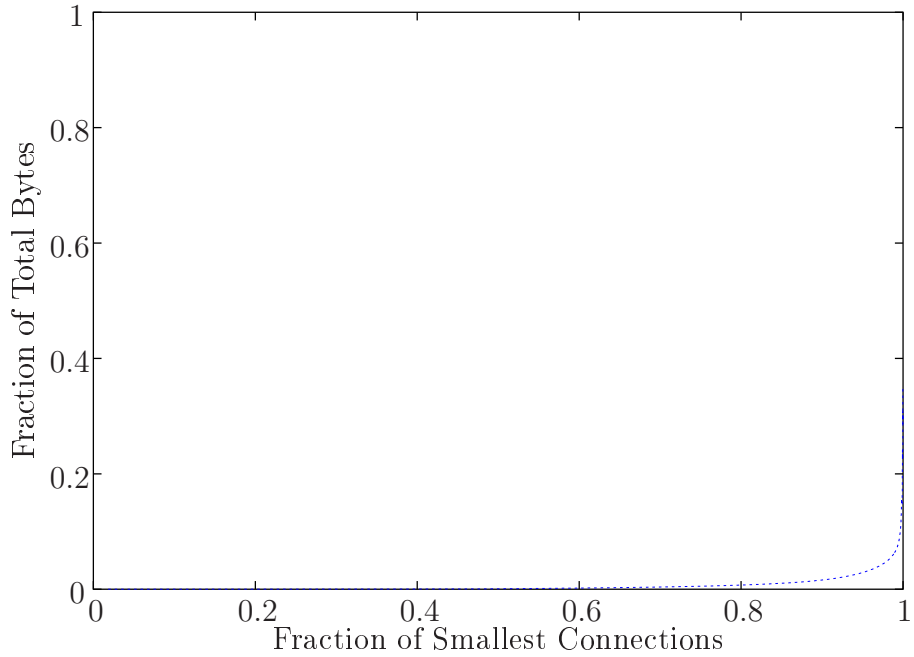


Figure 4.1: Trace Bytes as a Function of the Smallest Connections

An interesting property of heavy-tailed distributions is the mass-count disparity property. The intuition behind this property is that, when the size of processed

[4]More precisely, a random variable X is considered heavy-tailed when its cumulative distribution function $F_X(x)$ is $F_X(x) \sim 1 - cx^{-\alpha}$, where $0 < \alpha < 2$.

objects follows a heavy-tailed distribution, a large fraction of the bytes can be served by just taking care of a small fraction of the subsets [Crovella, 2001].

Shunting works as long as it is able to process most of the traffic in the hardware component. While the analyzer policy is the one that specifies traffic subsets, and how each one must be processed, the mechanism (the hardware component) must be able to process most of the traffic with a limited storage unit.

The key insight behind shunting is that a very large proportion of the traffic in a link is composed by a specific traffic subset, namely the set of high-volume connections. Moreover, from these connections, intrusion detection is interested only in their context, and not in the bulk data transmission.

4.4 Shunting

Inline traffic processing is a demanding activity. A passive packet-processing engine that drops packets just gets a reduced view of the traffic flow. While packet drops remain a small percentage of the total amount of packets, most packet-processing engines can bear dropping traffic without too many problems.

In comparison, inline packet-processing engines must inject back in the wire any packet that they capture. This provides an extremely powerful tool, as no packet passes the inline processing element without the element permitting it. On the other hand, the inline element dropping packets affects the quality of unreliable connections, and the throughput and latency of reliable connections. This effect is magnified by the

fact that packet losses are interpreted by TCP endhost stacks as congestion signals.

The remainder of this Section is structured as follows: Section 4.4.1 describes bottlenecks when processing high-speed traffic with off-the-shelf hosts. Section 4.4.2 presents the Shunting architecture, and Section 4.4.3 its rationale. Section 4.4.4 introduces the main processing mechanisms. Sections 4.4.5 and Section 4.4.6 discuss the architecture in depth. Section 4.4.7 describes some applications of Shunting. Finally, Section 4.4.8 compares Shunting to BPF-based approaches.

4.4.1 Inline Processing Bottleneck

A fully-used Gigabit link creates too large a workload for a commodity PC to do inline process. The problem presents different edges, depending on whether the NIDS is stateless or stateful. The former typically suffer because of the load imposed in the CPU, and the latter because of the state volume impact on memory and host bus interface bandwidth [Dreger et al., 2004].

Some back-of-the-envelope numbers may provide an idea of the magnitude of the bus capacity problem. NIDS need at least to (a) transfer the packet from the network interface to the host memory, (b) process the packet, and (c) inject it back to the interface.

In some cases, the packet may not be injected back into the network. This is the case, for example, in network intrusion prevention, when a packet is deemed malicious. In any case, in any network intrusion scenario, we expect malicious packets to be a

very small fraction of the total traffic, so the ratio of packets dropped should be negligible.

Moving the traffic from the network interface to the CPU and then back to the network interface for injection would account for 2 copies per packet, which assuming a (bidirectional) 1 Gbps link (2 Gbps of traffic), implies 4 Gbps of peripheral bus and memory bus traffic, considering no other traffic competes for the resources, and that the interrupt overhead on the bus does not limit the number of transactions per second.

If the NIDS is being run in userland, this means at least two extra data copies, to pass the packet from the kernel to userland, and back to the kernel. (Some Operating Systems add extra copies [Schneider, 2004].) This would means 2 more data copies, or another 4 Gbps in the memory bus.

Bus and memory are processing bottlenecks in stateful NIDS. A fast, conventional PCI (2.1) bus works at 66 MHz and has a 64 bit wide bus. Therefore, its maximum theoretical transfer rate is approximately the 4 Gbps. On the other hand, PCI is a shared bus, and transaction overhead (scheduling, addressing, and routing) plus contention and collisions typically reduce the effective bandwidth to one third of this [Arramreddy and Riley, 2002; Cleary et al., 2000], far away from the needed 4 Gbps.

This is changing fast. New parallel host bus architectures (PCI-X) are faster (PCI-X 2.0 at 533 MHz reports a 34 Gbps peak rate) and more efficient than the

previous models: Depending on the block size transmitted, PCI-X is able to provide between 66% and 85% of the theoretical bus bandwidth [Arramreddy and Riley, 2002; Compaq, 1999]. At the same time, high-speed serial interfaces such as PCI Express provide scalable, efficient performance (PCIe lines operating at 2.5 GHz have a maximum theoretical transfer rate of 3.2 Gbps per line) [Brewer and Sekel, 2004].

This is not the case for memory, however. Copying data between memory and the network adapter for a 1 Gbps link is already a difficult task, for both bandwidth and latency reasons. Problems are only expected to grow when processing packets in higher speed links (10 Gbps and 40 Gbps), as the memory gap keeps widening.[5]

On the other hand, these approaches still leave small headroom for the NIDS analysis, especially when considering (a) we have not considered the overhead caused by non-data touching processing [Kay and Pasquale, 1993], and (b) we want NIDS monitoring 10 Gbps or 40 Gbps links. The goal is for Shunting to provide NIDS with ample headroom to perform extensive analysis, instead of running on the edge and having to sacrifice it.

4.4.2 Shunting Presentation

Shunting is an architecture to perform packet processing on high-speed links. Figure 4.2 shows the Shunting architecture: A Shunt consists of two elements, a

[5]According to Gilder's Law [Aboba, 2001; Gilder, 2000], we should expect networking capacities to grow at least three times faster than CPU processing power (doubling every 9-12 months, as compared to every 18 months for CPU power, following Moore's Law). The "memory gap" is even bigger: Memory latency (increases only at 10% per year) and bandwidth increase even slower than CPU processing power [Patterson and Hennessy, 2004; Patterson et al., 1997].

software packet processing engine (the *shunt engine*), and a hardware active element (the *shunt device*). The shunt device elements can be thought of as a smart NIC. The shunt engine is the decision mechanism, and is composed of two parts, namely an external analyzer that processes the traffic (for example, a NIDS), and a thin layer that sits in the middle and knows how to make analyzer and device interact (the shunt *shim*).

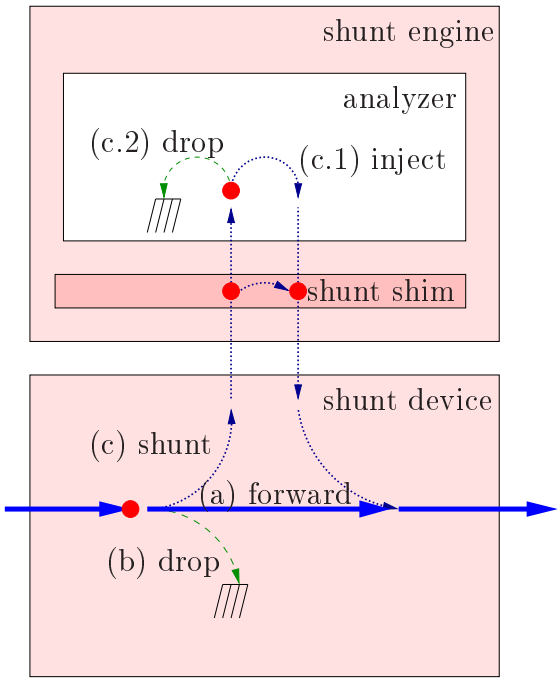


Figure 4.2: Shunting Main Architecture

When a packet arrives to the device, the latter has three possibilities: It can (a) forward the packet to the opposite interface (thick, solid line), (b) drop it (thin, dashed line), or (c) send it to the analyzer (thin, dotted line). The latter is also known as to *shunt* the packet.

In the intrusion detection scenario, for example, the three options can be mapped

to the device's judgment on the packet's goodness, namely whether the packet is innocuous (forward the packet), malicious (drop it), or none of the above (shunt it). The latter include packets cataloged as suspicious, or for which the device has no opinion.

Packets neither dropped nor forwarded are sent to the analyzer, where they can be processed more carefully. After this processing, the analyzer takes another decision about the packet's fate: It can (c.1) inject back the packet to the network interface, or (c.2) drop it. In the intrusion detection scenario, again, the analyzer can be a heavy-weight Intrusion Detection System, which makes another judgment on whether the packet is malicious (drop it) or innocuous (inject it back to the network interface).

There are several interesting points worth noting:

- Shunting is just a mechanism. The policy is decided by an external analyzer, which takes advantage of Shunting. In that sense, Shunting is policy neutral.

- The Shunting mechanism is extremely simple. The goal is to permit a very fast hardware implementation with lots of space for table entries.

- In normal operation, we should expect the large majority of traffic to be forwarded by the device, and therefore never reach the engine. This will permit to run the analyzer in commodity hardware.

- A shunted packet is delayed and processed by the analyzer. If the analyzer finds the packet correct, it forwards it. In the intrusion scenario, it means the device

cataloged it as suspicious, but the analyzer later acquitted it.

- Every packet is shunted by default, so it is the analyzer's obligation to instruct the device on which packets to forward, drop, or shunt. In the intrusion detection world, all packets are suspicious unless otherwise proven.

- Unlike the device, the analyzer need not take a decision as soon as it sees a packet. Instead, it can queue it, and process it later. In the intrusion detection scenario, if the packet is still suspicious, it can be stored until more information is available.

- Another possibility is "cache and shunt". The idea is that the device would cache the packet, map it to a unique identifier, and send a copy to the engine alongside the identifier. The latter would take a decision, and communicate it to the device as a {identifier, decision} tuple. This would save memory bandwidth in the engine to device path, as a packet injection would cause the engine to send just an identifier to the device, instead of the full packet contents.

  We chose not to implement this idea for simplicity's sake. One of the main goals of Shunting is to be able to implement the shunt device using fast hardware, where memory is a scarce resource. Holding packets in a hardware component would imply reducing the space for table entries. We therefore chose not to add it.

- We considered, but chose not to include, the possibility of the device reassembling

fragments before sending them to the analyzer. We decided against this idea, for the same reason as in the previous point (memory is a scarce resource in a hardware device.)

- There is an extra possibility we are studying, namely "forward and shunt": The device would forward the packet into the wire, and at the same time send a copy to the analyzer for processing. The latter would not need to reinject the packet back into the link, therefore saving bandwidth. In the intrusion prevention world, for example, this could be useful to monitor connections deemed non-malicious (for example, to know when they finish so that their associated state can be freed).

4.4.3 Rationale

In order to justify Shunting, we first make three simple observations that hold for some inline packet-processing applications, including intrusion prevention:

The first observation is that, while the per-packet processing may be very intensive (including deep analysis of the packet and others it is related to), the final decision on what to do with the packet is very simple: whether the packet must be forwarded or not.

This observation implies that only a generic host (the analyzer) is flexible enough to perform the potentially intensive work that permits deciding how to process a packet.

The second observation is that applications are often able to easily specify some subsets of the traffic for which (a) no analysis at all is required, and (b) the decision on what to do with a packet is the same for all the subset packets. These subsets include, for example, all packets from the same connection, and may be specified in advance or dynamically (i.e., as a consequence of the packet processing itself).

This observation implies that, once the generic processor is able to specify an easily processable subset of the traffic, processing that subset can be pushed into a specialized hardware element (the device).

The third observation relates to the heavy-tailed nature of network traffic (see Section 4.3.10). The number of bytes per connection has been shown to follow a heavy-tailed distribution [Crovella and Bestavros, 1996; Paxson, 1994; Paxson and Floyd, 1995]. We believe that the analyzer is able to take a decision on how to process the connections in the distribution tail (the very large ones) early in the connection life. Therefore, most of the connection contents will be processed only bu the hardware device. By focusing on the large connections, a limited number of entries in the hardware component are enough to process a large part of the bytes in the wire.

From these three observations, we draw the following hypothesis: In some scenarios, as intrusion detection, it is possible to do useful packet processing by dynamically specifying subsets of the traffic that (a) compose most of the traffic, (b) can be processed using simple, table-lookup operations implementable using a hardware

component.

Let's illustrate the hypothesis with an example.

Consider a NIDS monitoring a link. Every time it receives a packet, the NIDS must take a simple decision on what to do with it: either let it pass, or drop it. While the decision is simple, taking it may require an intensive task, including deep inspection of the packet, and querying information about previous packets from the same connection. On the other hand, sometimes the intrusion detection system is able to decide that any future packets from the same connection will be forwarded, without previous analysis. Then, it can instruct a hardware component to process those packets by itself.

4.4.4 Actions

Shunting works as long as most of the packets are processed only at the device, and the device operations remain simple enough to be efficiently implementable in hardware. The question is, therefore, which operations must be included in the device.

Some examples of useful operations in the intrusion detection scenario include dropping all packets related to hosts found to carry out attacks or port-scans, or forwarding all packets related to connections known to be safe.

Another example can be drawn from an accounting scenario, where the goal is to quantify the amount of traffic used by each connection [Mills et al., 1991]. In this case, relatively reliable accounting of TCP connections may be carried out by just shunting

the connection-establishment and connection-teardown handshakes, and forwarding all the remaining traffic.[6]

In order to provide a common framework where different analyzers can program their decisions, Shunting provides a series of "actions" that state, for every processed packet, whether it must be forwarded, dropped, or shunted. In order to combine differing actions, programmed actions are associated with a priority, and the action with the highest priority is followed.

In order to achieve the goal of analyzing most of the traffic using simple hardware processing, we have added actions only after considering (a) the benefit they provide to quick classification, and (b) the resource budget they consume from the other actions. While an extra action will always augment the flexibility of the device, it will consume part of the device's limited resources.

We have identified four actions that permit quick packet classification for most of the traffic in several important scenarios. These four actions are connection, address, port, and filter.

Connection Table

The first action that permits quick packet processing is the connection, considered as the traditional 5-tuple (104 bits) that defines a TCP connection ({source address, source port, destination address, destination port, transport-layer protocol}). With this action, connections can be matched to the 3-valued decision, namely forward,

[6]Section 2.5.2 in Chapter 2 discusses this accounting method in depth.

drop, or shunt. When the device receives a packet, it looks for its connection tuple in the connection table. If the connection tuple matches, the corresponding decision is used to process the packet.

The connection action is *easy* to implement. From the device point of view, it implies reading 104 fixed bits in every packet, then querying a table that yields a 3-valued decision. This ease permits implementing this table with a *very large number of entries*.

From the analyzer's point of view, the connection seems a natural category in which to take decisions. In the intrusion detection world, for example, it is easy to find cases where all packets from a connection are dealt with in the same fashion, dropping all of them if the connection is malicious, and forwarding all of them if the connection is safe.

The connection action is also *effective*, by leveraging the heavy-tailed nature of connection sizes and durations [Crovella and Bestavros, 1996; Paxson and Floyd, 1995]. If, in the general case, the analyzer is able to limit the processing of a connection to just a few of its initial packets, and then take a decision on how to process any additional packets from the same connection, it is possible to diminish the total amount of traffic the analyzer processes, while effectively processing all connections. This seems a useful aid in any packet processing application that requires flow state management, as intrusion detection or flow classification.

This action can also be useful to limit the traffic load a server must bear. If

the analyzer detects an unexpected, non-malicious spike in the amount of traffic directed to a server that may overwhelm it, the analyzer can opt between (a) to keep monitoring all the traffic it sees, but simplifying the monitoring type, and therefore reducing the amount of per-packet processing; or (b) to keep the same amount of per-packet processing, but limiting the total amount of monitored traffic, for example, by forwarding some of the connections without further analysis. This is a key policy decision that should be taken by the analyzer, not by the Shunting architecture.

Note that the insertion of a *forward* tuple in the connection table is caused by the analyzer guessing that further processing is not needed, or that further processing is not possible. The latter may happen, for example, if the analyzer knows it does not know how to analyze a connection (because, say, it is an unknown protocol). In this case, it may just forward all such traffic.

Let's discuss an usage example. Consider a NIPS monitoring an SSH connection. An SSH connection consists of two parts [Ylonen, 1996]: First, a clear-text, session handshake, in which the client and server (a) exchange identifier strings, (b) agree on the ciphers and authentication method they will use, and (c) create the session key that will be used for the rest of the session. Second, encrypted traffic, in which the application data is exchanged.

The first packets of the connection are shunted to the NIPS. After the clear-text session handshake ends, the software NIPS may realize that the connection is dangerous, for example if any of the SSH version identifier strings shows the SSH server is running

a buggy software version. If the NIPS realizes this, any additional packet belonging to the connection can be dropped without further packet analysis.

Conversely, if at any moment the NIPS deems the connection is safe, any additional packet belonging to the connection can be forwarded without further packet analysis.[7] Moreover, and assuming that the NIPS does not know the session key, the NIPS cannot peek into the connection contents once they get encrypted. Any further packet analysis is therefore a resource waste. If the NIPS did not find anything bad in the connection, it will not be able to do so in the future. Therefore, it can just pass along any additional packet from the connection, without inspecting it.

Address Table

The second action that permits quick packet processing is the IP address, of both the source and the destination. IP addresses can be matched to 3-valued decisions, namely forward, drop, or shunt. When the device receives a packet, it reads both its source and destination addresses, and looks for them in the address table. If the address tuple matches, the corresponding decision is used to process the packet.

The address action is *easy* to implement. From the device point of view, it implies reading 64 fixed bits in every packet (32 per address), and then carrying out two queries to a table that yields a 3-valued decision. From the analyzer point of view, the address seems a natural category in which to take decisions. In some cases, it

[7] The analysis actually may want to wait a little bit to guess if the client is using brute force to log in the server. This is typically seen as three tries and fails in the encrypted stream, and can be guessed by checking the packet sizes before termination.

is easy for the analyzer to develop a simple drop/forward policy for all the traffic coming from or going to a given host. This is also known as host blacklisting and host whitelisting.

The address action is also *effective*. We believe an important percentage of the traffic may be processed by just looking at each packet source and/or destination addresses. For example, if a NIPS detects that an external host is scanning the network, it may decide to blacklist it. This means that any further packets whose source address is that of the scanner will be dropped without further processing.

The opposite may be also true. Some organizations use a host to scan their network in search of vulnerabilities. These scanners look at the internal network hosts for services that are misconfigured, outdated, or just plainly forbidden. A well-known mechanism to carry out this search is to open connections to all the ports where such services can be located. In most cases, the service will not exist, and the connection will fail.

Of course, this behavior is not that different from that of port-scanners, so a network analyzer may qualify the scanner's traffic as malicious. In this case, the NIPS must not drop the scanner traffic. What's more, it may decide all traffic related to this type of host is innocuous (albeit it looks malicious), and therefore whitelist it.

The address action also permits Shunting to defend hosts inside the network against unusual surges of malicious traffic (flooding), by temporarily disabling their accessibility from the outside network. Consider a distributed flooding of an internal

host. If the engine detects the amount of traffic directed to a host increases excessively, but such traffic has anomalous characteristics (for example, a large asymmetry, or an unusual percentage of connections closed just after the initial SYN), it can instruct the device to drop any packet directed to the attacked host, effectively isolating it from outside traffic.

Note that the isolation can be further refined to permit host-initiated connections to be forwarded, by using *priorities* (see Section 4.4.5).

Shunting can use this isolation technique to defend itself from such floodings. While most attacks recorded so far against intrusion detection and prevention systems are based on software bugs in such systems [Moore and Shannon, 2004], there are already some tools that attack the analyzer itself, either by increasing the number of false positive, or by increasing the workload until the analyzer gets overwhelmed. Some of these tools include Stick [Giovanni, 2001], Squealing [Patton et al., 2001], and Snot.

Last, the isolation technique provides a mechanism for the analyzer to defend itself against unusual surges of well-behaving traffic: The ability to forward packets without any in-host processing. If the analyzer detect some large increase in the amount of traffic it is processing, it has the option to instruct the device to just forward a subset of it. This would invalidate the analyzer's protection in the traffic subset, at the benefit of keeping the protection in the remaining traffic. Again, this is a policy decision the Shunting mechanism is neutral to.

Port Table

The third action that permits quick packet processing is the transport protocol port, including both the source and the destination one. Transport (TCP or UDP) ports are matched to 3-valued decisions, namely forward, drop, or shunt. When the device receives a packet, it reads both its source and destination port, and looks for them into the port table. If any port tuple matches, the corresponding decision is used to process the packet.

The port action is extremely *easy* to implement. From the device point of view, it implies reading 40 fixed bits (the two ports plus the transport protocol identifier) from every packet, and then carrying out two queries to a table that yields a 3-valued decision.

From the engine point of view, the port seems a natural category in which to take decisions. There are some ports that correspond to well-known insecure services (telnet), and others which, while not being insecure, its presence in a DMZ is rarely justified (e.g. Microsoft NetBIOS or NFS traffic). If the site policy considers that traffic in any of these ports constitutes a security problem, the port table mechanism permits dropping all their packets. Shunting permits refining this policy by using several tables at the same time. (See Section 4.4.5 for a discussion on the priority system.)

In the intrusion detection world, the port action can be used to slow the spreading of fast worms. If the analyzer detects an extreme increase in similar traffic directed

to an unusual port, and it concludes it is due to worm activity, it can instruct the device to drop all traffic directed to that port. On the other hand, we doubt that we can ever set an entry in the port table to forward all traffic corresponding to a specific port. That could be used by an attacker: by setting her traffic local port to be equal to the forwarded port, she would in fact launder all her traffic, getting a free pass on the NIPS. Directionality would help here, as we will see in Section 4.4.5.

In the accounting scenario, on the other hand, we imagine there are some protocols that, because of policy reasons, are not interesting. These ports could be dealt with by setting an entry in the port table to *forward*.

Note that, while we understand port and service are independent entities, we are assuming they are normally related, at least in the local network endhost, which is the side the intrusion detector is trying to defend.

The utility of the port table is definitely more limited than the two previous tables. On the other hand, the cost of a port table is very low, as a 3-valued yield requires 2 bits per port. Considering the two main transport protocols, TCP and UDP, every protocol has 64 K ports. This implies we can carry information about all TCP and UDP ports for just 32 KB of memory.

Filter

The last action that permits quick packet processing is three fixed BPF filters, known as the "forward filter," the "drop filter," and the "shunt filter," that cause

packets matching them to be always processed the same way (forwarded, dropped, or shunted, respectively).

Note that the three filters are static. The option of implementing a generic BPF engine in the device, where dynamic filters could be uploaded at will, is complicated in hardware terms, and would limit the hardware parallelization opportunities.

The static nature of the three filters is compensated by combining their use with the dynamic tables. In this way, we envision that they could be useful in helping the analyzer managing its state.

As an example, while an intrusion detection system may decide to forward all packets from a given TCP connection, it would benefit from seeing the connection termination segments, as it can free the associated space. In a similar fashion, an accounting analyzer could benefit from seeing such termination segments, from which it could estimate the total connection size.

In order to ensure the connection termination segments are always sent to the analyzer, while at the same time forwarding a connection's bulk transmitted data, a plausible approach could be to set the static shunt filter so that TCP segments with the RST or FIN flags set match, and then add, with a smaller priority, an entry to the connection table causing packets from the connection to be forwarded.

A packet belonging to such connection will be directly forwarded, except when it has the RST or FIN flags set, in which case it will be shunted.

The static nature of the three filters makes them a likely target for attackers. For

example, if an attacker wants to flood a NIPS which she knows uses a "TCP RST or FIN segments" shunt filter, she could send a large amount of TCP segments with any of the two flags set. If the attacker can be identified, and she is not spoofing the IP source address, the attack can be thwarted by setting an even higher-priority entry in the address table, associating the attacker address to a drop result.

The filter action is easy to implement, considering it is static. For example, the shunt filter mentioned before ("TCP RST or FIN segments") can be implemented by reading just the network and transport protocol header.

We expect the filter action to be effective, not on the amount of traffic it will be able to uniquely describe, but in the effect that such traffic will have in the engine (state management). Moreover, we expect the filter to be simple enough so as to impose a very low burden on the device. In particular, its memory usage will be very small.

4.4.5 Other Details

Some details about how the shunt architecture works are:

- Priorities: The shunt architecture includes the idea of fine-grained (per-tuple) priorities, associated with different actions. The objective is twofold. First, it is intended to solve potential conflicts between decisions obtained from different actions. For example, let's assume the port table has an entry that states that packets from a given port are to be forwarded. Let's also assume that the

address table has an entry that states that packets directed to a given host are to be drop. If a packet matches both actions, the device will follow that with the highest priority.

Second, fine-grained priorities permit the user to have at her disposal a hierarchical decision system. Local, low-priority defaults can be set with the protection of higher-priority safeguards. An example is how to instruct the device to (a) forward all packets from a given TCP connection, while at the same time ensuring (b) the connection teardown segments are sent to the engine, and (c) an attacker cannot overwhelm the engine by sending lots of TCP control segments.

As we mentioned before, priorities may be used to address this problem: A packet belonging to the given connection is forwarded with low priority, a packet with any TCP control flag on is shunted with medium priority, and a packet from a well-known attacker is dropped with high priority. The device, therefore, checks every packet with the four actions, and takes the highest priority decision.

We define a *conflicting match* as the case where two tables provide different actions with the same priority. This is considered an error, and the result is that the packet will be shunted.

Note that priorities are set in a per-tuple basis. Therefore, different tuples in the same table may have different priorities.

- Directionality: Another added point is the influence of directionality in the

connection, address, and port tables. A connection table tuple matches packets going in both the forth and back directions. Different yields can be set for the same connection. For example, you can set packets going in the forth connection to be forwarded, while packets going in the back direction are shunted. This way, the engine will only see one side of the connection.

For the address table, directionality helps differentiate source and destination port. Again, it makes perfect sense for the device to take different decisions depending on whether the packet goes to a given host, or comes from a given host. The same reasoning can be applied to the port table.

- Default Shunting: If no actions matches a packet, it is shunted to the engine. The main goal of this decision is analysis exhaustivity, i.e., ensuring that any packet whose processing has not been made explicit in advance, will be sent to the engine.

  Default shunting introduces two interesting concepts: First, "safe decision", i.e., shunting, which is never wrong in functionality terms (though it may be in performance terms). This presents an additional advantage: It permits hardware designs where, in order to increase efficiency (in performance or space terms), results are only probabilistic or even knowingly incorrect.

  Shunting supports such implementations while their errors are single-sided, i.e., they change *forward* and *drop* into *shunt*, but they never transform a *shunt* into

a *forward* or a *drop*.

An example of the usefulness of a hardware design that may knowingly produce incorrect results is devices with limited storage space. In these devices, table entries can be evicted for space reasons. Consider the case of a packet that would have been matched by a tuple in one of the tables, but such tuple has been evicted due to lack of space. As no tuple exists now, the packet will be shunted by default. If the original decision was *shunt*, the packet will go the right path. If it was *forward* or *drop*, it will go the wrong path (it will be shunted), but once it arrives to the analyzer, the latter will fix its path. The engine will then either inject the packet back in the network interface, or drop the packet itself (and probably update the corresponding table). The final packet processing will be the same in the limited storage device as in a device with unlimited storage space, at the cost of processing a packet through the shim instead of just in the device.

The second concept is the possibility of the shim reissuing entries that have been evicted by the device. If the device, because of limited space, needs to evict an entry in one of the tables, the shim will know, as it will get a packet that should have been processed directly by the device. In this case, the shim may reissue the entry into the device table, therefore optimizing the composition of the device tables.

The main problem of default shunting is cold-start. When the shunting system

is started, it is swamped with the full link traffic. This high-load situation lasts until the engine manages to populate the connection, address, and port tables. Moreover, some of the traffic corresponds to partial connections, i.e., connections from which the engine will never see the start. The analyzer in the engine should be able to take policy decisions based on such partial connections. These decisions cannot just be forward all partial connections: This could be taken advantage of by an attacker: She could launder all her TCP traffic by sending a non-SYN segment to the victim address and port before the real connection. The analyzer would think it is a partial connection, and forward any further packet.

There is also another problem related to Default Shunting, which is discussed in Section 4.7.3.

- Sampling: Another functionality added to the Shunting architecture is the introduction of per-tuple, random sampling. Alongside the forward/drop/shunt yield (in both directions) and the priority, all entries in the three tables and all static filters have a shunt sampling ratio. When a tuple's yield is *forward* or *drop*, or either the *forward* or the *drop* filter match a packet, the corresponding shunt sampling ratio is checked. If it is not zero, a random decision with the mentioned sampling ratio is taken. If the decision is sample, then the *forward* or *drop* yield is substituted with a *shunt* yield, and the priority is kept the same.

Sampling permits the analyzer to carry out tricks like receiving small pieces of a large subset of traffic without having to capture the full subset. An example case is a large connection that the engine cannot afford to process, and therefore sets to *forward*. On the other hand, the engine is willing to see a packet from time to time, in order to detect strong bitrate variations, or absence of activity.

Note that the sampling infrastructure has been designed to be very cheap. One of the main architecture concerns is space, and there is a sampling ratio for each table entry. We do not want to spend 32 bits just for each sampling ratio. Instead, we have limited the sampling rate representation to a few bits of granularity, 3 in our primary implementation. This permits 7 different sampling ratios, plus 0 for "no sampling." The exact meaning of each of the 7 sampling ratios can be defined by the analyzer.

- Interconnection: An important implementation decision is the interconnection between the shim and the shunt device. Because we want the analyzer to have access to generic processor capabilities, the shunt engine will run in a generic host. As for the device, we have considered two different options.

  First, the device may be interconnected using the host local bus (for example, PCI). This presents simplicity advantages: Installing a shunting host should be as easy as plugging a new card in the local bus, and the shim and the device would not need care about communication reliability.

Second, the device may be interconnected using a generic network link (for example, Gigabit Ethernet). The main advantage of this approach is that it permits decoupling device and engine physically, and because Ethernet is a shared medium, to combine several devices with several engines.

This presents several interesting possibilities, like a single engine managing several devices, which could be used to, for example, achieve a centralized, complete view of the DMZs linking a domain to the rest of the internet; or a single device dividing its load among several engines, so that the workload can be shared between the analyzers in the different engines; or a combination of both.

The architecture is neutral to the interconnection flavor.

Figure 4.3 shows an abstract representation of the shunting decision process. A packet is passed through the four actions (the three tables plus the static filters), and each action produces a tuple {decision, priority}. The decision taken by the device is the one corresponding to the yield with the highest priority. (If a table does not contain the packet's corresponding entry, or if none of the three static filters matches the packet, then the corresponding tuple contains a *shunt* decision with minimum priority.)

Figure 4.4 shows the structure of the three tables. Note that the yield occupies only 10 bits per entry, including 4 for the forward and shunt action in both directions, 3 for the priority field, and 3 for the sampling capability.

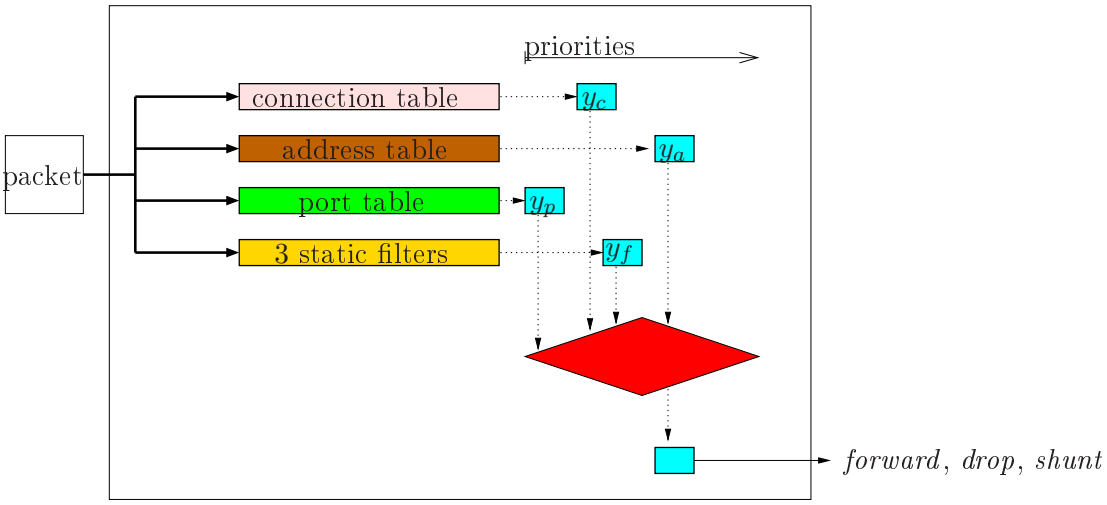


Figure 4.3: Shunting Decision Process

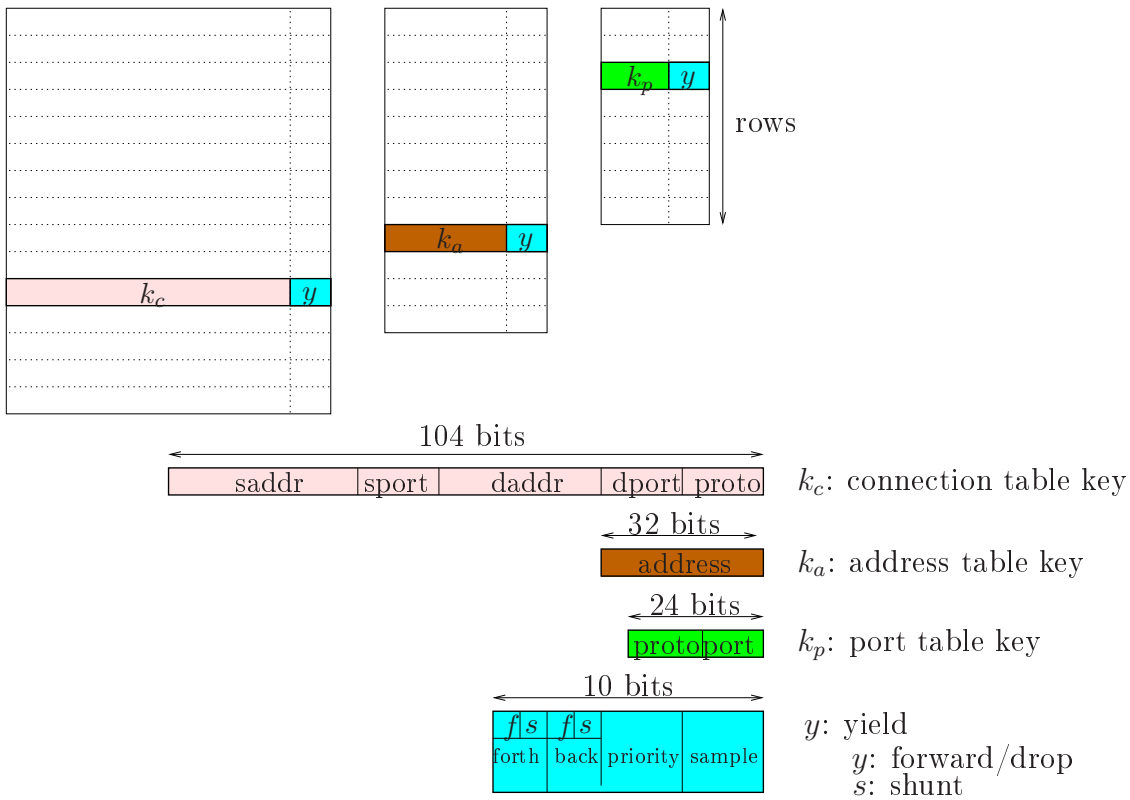


Figure 4.4: Shunting Tables

4.4.6 Discussion

The rationale behind the proposed mechanism is that the engine can typically state which traffic it need not process, or cannot process, and that it can express its statement in the aforementioned actions. Therefore, the engine will only receive traffic it can and must process.

The performance benefit of the Shunting architecture relies in processing the majority of the traffic at the earliest possible stage, the device. Most of the traffic never goes into the host, therefore saving the host's scarce resources (bus, memory, and CPU).

In order to do this processing, Shunting uses the shunt device, an augmented NIC that permits generic packet processing offloading. As any hardware element, the device sacrifices flexibility for performance. It can only perform a reduced set of actions (forwarding, dropping, and shunting packets) based on four simple actions: connection, address, and port tables, and filter. In exchange, the device can perform very fast packet processing.

To control the device, Shunting uses the shunt engine. The device provides programmable functionality to classify traffic. The use of that functionality is decided by the engine. The device is the mechanism, and the engine provides the policy.

This simple setup permits reducing the amount of processing in the software engine, as it can offload processing to the device, while at the same time allowing for fast processing in the hardware device, as the simplicity of the four actions allows

for fast implementations.

Another benefit of Shunting is to permit engine self-defense. This self-defense can be actually weighted with the current engine workload. For example, when feeling overwhelmed, the engine may decide to forward some subset of the traffic because it is low-value to process versus the load it requires.

Yet another benefit of Shunting is to provide a framework where to permit efficient analysis of traffic in non-standard ports. The analyzer could fingerprint any connection it sees, and once it manages to do so, it could avoid seeing any more packets from such connection by inserting a corresponding entry in the connection table.

4.4.7 Applications

This section describes some applications of the Shunting architecture.

- The main Shunting scenario use we devise, and the one evaluated in this Chapter, is network monitoring. This includes both network intrusion detection [Paxson, 1999] and network debugging [Agarwal et al., 2003]. For network monitoring, Shunting can be seen as a flexible, though efficient, Input-Volume Control Technique (see Section 4.3.5). Shunting allows for dynamically deciding which packets reach the engine. This provides a very powerful tool to reduce the amount of state created in intrusion detection systems, and to efficiently capture traffic in network debugging systems.

- Shunting also permits performing network monitoring in subsets of traffic that

require incremental, dynamic filter specifications. Some examples of data sessions being dynamically negotiated include: (a) multimedia streaming, where the bulk media data is sent over a UDP session (RTP, session control channels) that is negotiated during a control protocol session (RTSP, H.323) located at a well-known TCP port [van der Merwe et al., 2000]; (b) capturing FTP data in either active or passive transfer mode, where the bulk data transmission is done using a connection defined during the FTP control protocol [Postel and Reynolds, 1985]; and (c) peer-to-peer sessions, where the full flow specification keeps changing through the life of the data transmission, when data providers (transmitters) keep appearing and disappearing.

- Another scenario where Shunting is useful is to perform load balancing in packet processors. The basic idea is to use shunting as very-fast packet capture devices for the packet processors. The idea of using a complex device (the shunting architecture, in our case) as a fast packet capture device is similar in spirit to [Deri, 2003]. In that case, the author proposes using a router. By setting the devices to divide the traffic using any of the three connection, address, and/or port tables, several packet processors can be limited to process only a subset of the traffic. The main problem of placing Shunting systems in parallel is that it can increase packet delay and packet reordering.

- Shunting can also prove useful for traffic accounting. Shunting helps to capture

information about connections without having to pass the full data through the analyzer. Also, efficient sampling permits probabilistic billing.

- As we discussed in Section 4.3.4, Shunting can be used to perform traffic normalization.

4.4.8 Comparison with BPF-Based Approaches

This Section enumerates the problems of performing inline high-speed packet processing using only software and/or hardware approaches based on the popular BPF packet filter architecture [McCanne and Jacobson, 1993].

In-host BPF, i.e., running the packet processor on a host, and using the kernel BPF implementation to decide which packets reach the processor and which do not, is not viable in high-volume environments, as all packets will reach the host, even if they are just forwarded. This means that the host must be able to bear at least two times the bitrate of the link it is processing.

Pushing BPF to the NIC, while solving the host bus and memory bottleneck, presents three drawbacks, namely (a) it does not provide enough functionality, (b) it does not scale, and (c) it is too static.

In some ways, the semantic model of the BPF language is higher than that of the shunting architecture: While the total number does not exceed a few tens, any combination of connections, addresses, and ports can be specified using the BPF language.

Shunting provides simpler, fixed semantics with dynamic tables. The benefits are:

- BPF does not scale. In the in-host BPF case, a simple filter composed of several hundreds of primitives takes several minutes to compile, and runs extremely slowly, as the BPF engine must process all primitives sequentially. In the shunt device, we expect to have almost unlimited space for the address and port tables, and a very high number of entries for the connection table.

  The shunting architecture's goal, on the other hand, is to have tables with maybe a million connection entries.

- BPF filters are too static. Adding or deleting an address from the address blacklist in BPF requires recompiling the full filter and changing the full filter in the period of time between the processing of two packets. In the Shunting architecture, it only requires adding or deleting an entry in the address table.

- BPF is a packet capture filter, and therefore it produces binary decisions (capture a packet or not). Shunting produces multivalued decisions (forward, drop, or shunt).

- BPF is not as flexible as the whole shunt architecture, which uses a generic processor as the analyzer. As an example, byte-string signature detection in an efficient and generic way requires a richer syntax than that of BPF, including generic inter-packet state availability.

4.5 Design and Implementation

This Section describes the design of an implementation of Shunting, and its application to carry out network intrusion prevention.

This Section is organized as follows: Section 4.5.1 describes our implementation of a NIPS, based on integrating a NIDS with Shunting. Section 4.5.2 describes the communication between the the shim and the device, including the network API. Section 4.5.3 describes the shunt device. Section 4.5.4 describes the communication between the analyzer and the shim (the mechanisms and policies in the analyzer to drive the shim). Section 4.5.5 introduces the shunt shim, the thin layer that permits communication between the analyzer and the shunt device. Section 4.5.6 describes an example of the modifications required by an analyzer (Bro) to work with the shunting architecture.

4.5.1 Implementation Description

Figure 4.5 shows the implementation of a NIPS based on integrating a popular NIDS (Bro) and the Shunting architecture. Data traffic is shown using solid lines, while control traffic is shown using dotted lines.

The Shunting system is composed of two parts: a simple, hardware front-end ("shunt device"), and a flexible software component ("shunt engine"), which can run on an off-the-shelf host. The engine is divided into the analyzer (for which we use a modified version of Bro [Paxson, 1999]), and the shunt shim, which serves as glue

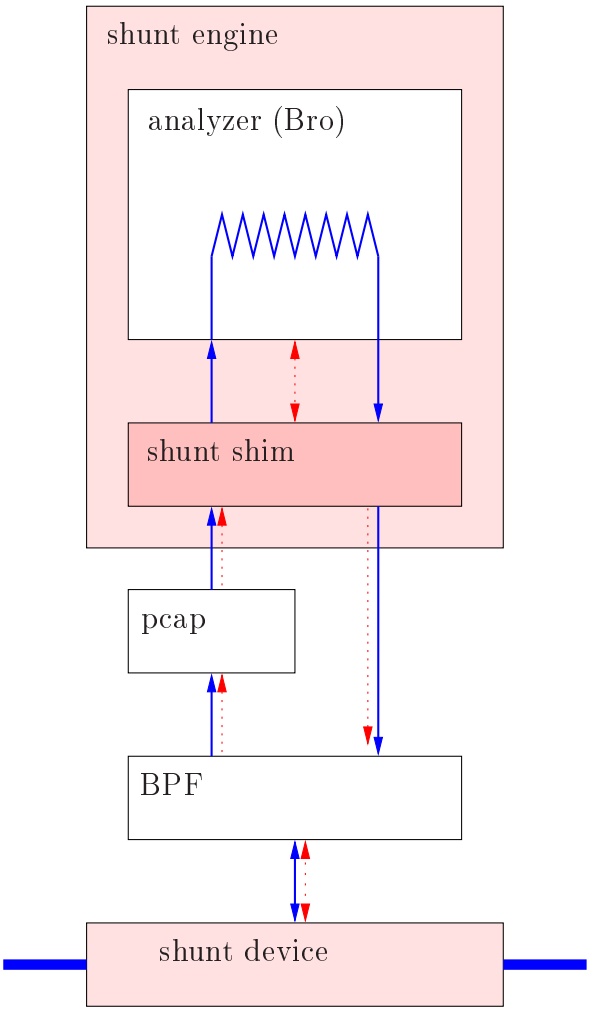


Figure 4.5: Design of an Intrusion Prevention System Using Shunting

between device and analyzer.

During the normal data operation (solid lines in Figure 4.5), the device receives packets through any of its two network interfaces. These packets may be forwarded to the opposite interface, dropped, or shunted. In the last case, packets are sent to the engine, which captures them using the standard libpcap over BPF mechanism.

In the engine, packets are received by the shunt shim and filtered again. The goal is to take into account the case where the shunt device made a mistake. If the packet gets shunted again, it is sent to the analyzer. Otherwise, it is either reinjected into the device (if the right decision would have been *forward*), or dropped.

The analyzer processes the packet. This processing may result in insertion or deletion of table entries in the shim and device (see dotted lines in Figure 4.5), for which the analyzer has been extended.

When the packet processing finishes, the analyzer re-injects the packet into the shim. The shunt shim performs a third filtering, this time to take into account the case where the analyzer modified a table in a way that changes the way the packet gets processed. If the result is again different from *drop*, the packet is sent back into the device, where it is finally reinjected back into the network.

4.5.2 Device-to-Shim Connection

In the current implementation, from the two options to make the device communicate with the shim (PCI bus and Ethernet connection), we have selected the latter for

this implementation. The reason is twofold: First, during the design and testing phase of the project, it is easier to simulate the the Ethernet connection (by using virtual devices) than the PCI bus. Second, the software device simulator used for debugging provides the full hardware device functionality (except the performance), and therefore can be used in cases where the processed traffic stream is intermediate, meaning high enough as to cause problems to an analyzer capturing packets directly from the network, but not as high as to require the real, hardware shunt device.

For the first implementation, the Ethernet connection used is point-to-point, instead of shared.

As the device and shim communicate using a dedicated Ethernet connection, they use network traffic to exchange information. This network traffic can be divided in data packets and control packets.

Data packets correspond to packets that are shunted (device to shim) and packets that are re-injected by the device (shim to device). Data packets are attached to some information on the decision taken. For example, when the device sends a shunted packet to the shim, it must attach information on why the packet was shunted (including the table that matched, and sampling information, if it applies), and the incoming network interface. When the shim sends back the packet, it must include the incoming interface, so that the packet can be re-injected in the opposite interface.

Control packets are sent using a new protocol, known as the Shunt Interconnect Protocol (*SHIP*), which runs over UDP.

Shunt Interconnect Protocol

Figure 4.6 shows the SHIP packet format. The shaded section represents the packet header, and the blank section represents the packet payload.

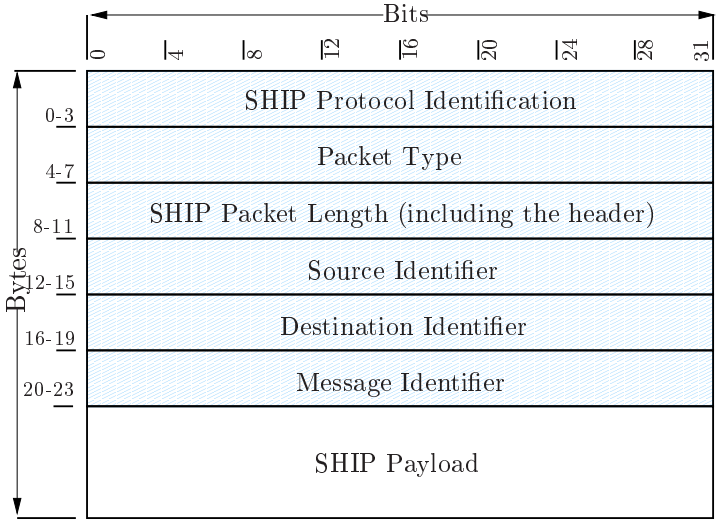


Figure 4.6: Shunt Interconnect Protocol Packet Format

Table 4.2 describes the 24 byte header fields.

field	length (bytes)	explanation
protocol version	4	support for protocol versions
type	4	control message type
length	4	SHIP packet length, including 24-byte header
source ID	4	support for multiple engines/devices
destination ID	4	support for multiple engines/devices
message ID	4	identifier to implement reliability

Table 4.2: Shunt Interconnect Protocol Header

The control message types, and the corresponding SHIP payloads, are described in Table 4.3. In this table, the "direction" information states whether the control packet can be seen in the shim-to-device direction ($s \rightarrow d$), device-to-shim direction ($s \leftarrow d$), or both ($s \leftrightarrow d$). The exact semantics and payload contents are described

in Section B.1 of Appendix B.

SHIP Reliability

If the shim and the device communicate through a network link, we must consider the possibility of packet losses, especially because the shared character of Ethernet presents issues like collisions, and physical problems which increase the loss rate from the channel Bit Error Rate (BER).

Our reliability approach uses a simple sliding window mechanism, and is used only with control packets, where losing a single packet may affect the shunt processing. For data packets, we assume the transport protocol in the corresponding connection end-hosts will care about reliability, if needed at all.

When either the shunt device or the shunt shim want to send a reliable control packet, they prepend a unique, consecutive identifier to the packet, send it, and store it in a fixed-size buffer. When either the device or the shim receive a packet marked as reliable, they acknowledge it by sending an ACK message back associated to the packet identifier.

Both sides use timers to achieve reliability. When sending a reliable packet at time t, the message sender also sets a retransmission timer at time $t + \Delta$ in the future, where Δ is a configurable, fixed value. On receiving an acknowledgment packet for the corresponding identifier, the buffer frame where the packet was stored is freed, and the retransmission timer is either canceled or reset. If, on the other hand, the

Table 4.3: Shunt Interconnect Protocol Payload

control message type (direction)	**parameters**	**explanation**
ack ($s \leftrightarrow d$)		message was received
	acked_msg_id	acked messaged identifier
device_ready ($s \leftarrow d$)		device is alive
	dev_id	device identifier
open ($s \rightarrow d$)		initialize the device
	filter_strings	forward, drop, and shunt filters
	filter_priority	forward, drop, and shunt filter priorities
	filter_sample	sampling ratio for the filters
	default_sample	default sampling ratio
	failsafe_mode	fail-safe mode (fail-open or fail-close)
capabilities ($s \leftarrow d$)		device capabilities
	version	device version
	nics	information on the network adapters the device listens to
close ($s \rightarrow d$)		close the device
close ($s \leftarrow d$)		the device had to close
reset ($s \rightarrow d$)		reset the device
	hard	whether the reset must be hard or soft
	stats	whether the device must reset its statistics
	tables	whether the device must reset its tables
error ($s \leftarrow d$)		report an error
	code	error code
status_request ($s \rightarrow d$)		request the device status
	on_off	send status periodically
status_response ($s \leftarrow d$)		return the device status
	data	device status (table contents)
statistics_request ($s \rightarrow d$)		request the device statistics
	on_off	send statistics periodically

Continued on next page

control message type (direction)	**parameters**	**explanation**
statistics_response ($s \leftarrow d$)		return the device statistics
	data	device statistics (information on packet/bytes forwarded/dropped/shunted/injected during last second/since start)
associate_conn ($s \rightarrow d$)		insert a tuple in the connection table
	src_addr	connection's source address
	src_port	connection's source port
	dst_addr	connection's destination address
	dst_port	connection's destination port
	forth_action	whether packets in the forth direction must be forwarded/dropped/shunted
	back_action	whether packets in the back direction must be forwarded/dropped/shunted
	priority	tuple priority
	sampling	tuple sampling ratio
associate_addr ($s \rightarrow d$)		insert a tuple in the address table
	address	the address
	forth_action	whether packets in the forth direction must be forwarded/dropped/shunted
	back_action	whether packets in the back direction must be forwarded/dropped/shunted
	priority	tuple priority
	sampling	tuple sampling ratio
associate_port ($s \rightarrow d$)		insert a tuple in the port table
	port	the port
	forth_action	whether packets in the forth direction must be forwarded/dropped/shunted
	back_action	whether packets in the back direction must be forwarded/dropped/shunted
	priority	tuple priority
	sampling	tuple sampling ratio

Continued on next page

control message type (direction)	**parameters**	**explanation**
deassociate_conn ($s \rightarrow d$)		deassociate a tuple from the connection table
	src_addr	connection's source address
	src_port	connection's source port
	dst_addr	connection's destination address
	dst_port	connection's destination port
deassociate_addr ($s \rightarrow d$)		deassociate a tuple from the address table
	address	the address
deassociate_port ($s \rightarrow d$)		deassociate a tuple from the port table
	port	the port
eviction_conn ($s \leftarrow d$)		a tuple was evicted from the connection table
	src_addr	connection's source address
	src_port	connection's source port
	dst_addr	connection's destination address
	dst_port	connection's destination port
eviction_addr ($s \leftarrow d$)		a tuple was evicted from the address table
	address	the address
eviction_port ($s \leftarrow d$)		a tuple was evicted from the port table
	port	the port

timer expires before receiving the corresponding ACK, the packet whose transmission time has also expired is retransmitted, and the associated timer reinstated.

In order to make things as simple as possible for the hardware device, ACK messages are individual, not cumulative, and they are assumed to be unreliable. This means that retransmissions only occur when a packet has not been acknowledged after Δ seconds. On the other hand, an ACK may also be lost. In this case, the timer corresponding to the unacknowledged packet will expire, and the packet will be sent again.

If, at any time, the number of retransmissions for a single packet sent from the device reaches a fixed threshold, or the device tries to send a reliable packet and the buffer is full, we assume the device and the shim have communication problems, and move the former to fail-safe state. The device drains completely its buffer, and sends two messages to the shim: One to report it had a buffer error, and the second to report it is ready to be initialized again.

If, at any time, the shim tries to send a reliable packet, and the buffer is full, the buffer is drained and a "full buffer" error message is sent to the shunt shim. When receiving this message, the shim knows something bad is going on with the device, and may opt for putting it in fail-safe mode, or resynchronizing the table contents.

Table Synchronization

Both the shim and the device have their own copies of the three dynamic tables.

The contents of the device tables need not be exactly the same than those of the shim tables. Instead, they will be just a subset of the contents of the shim tables.

The reason of this difference is that the shunt device is implemented in hardware, and therefore its capacity will be limited. In comparison, the shim runs on an off-the-shelf host, and therefore it may use more resources for the tables.

In some sense, the device tables work as a cache of the shim tables: They have less space, but in exchange they operate faster. The only requirement is that there are no entries in the device tables that are not in the corresponding shim table. Default shunting combined with the fact that the device is obliged to report any table eviction, ensures that the limited capacity of the device tables only affects the performance of the shunting process, not the correctness.

SHIP Synchronization

SHIP includes a mechanism to synchronize the tables device and shim. The synchronization mechanism is very simple: it permits sending the full table contents to the other side in an efficient fashion (several tables per packet, instead of one tuple per packet, as in the normal method).

The synchronization mechanism uses reliable transmission. In order to avoid using all the buffer retransmission space, only a handful of packets are sent at the same time. Further packets are clocked by incoming packets.

Synchronization works in both directions, i.e., synchronizing the shim with the

device, or the device with the shim.

In the first case, the shim sends all the entries in its three tables to the device, which uses them to fill its own tables. This synchronization method is useful after the shim moves the device from fail-safe state to working state. If the shim wants to populate the device quickly, using the normal table access mechanism will imply sending one SHIP packet per tuple. Instead of that, the SHIP synchronization mechanism permits inserting several tuples per packet (as many as fit in a network packet).

The device may not be able to fit all the entries in its tables, for example for lack of capacity. In this case, it may evict the entries that do not fit, provided that it reports the evictions to the shim.

In the second case, a useful scenario consists of the shim willing to know the contents of the device tables. This is useful, for example, when the shim and the device run on separate places, and the former crashes. When it is restarted, it may ask the device for its table contents. At the shim request, the device will respond by sending synchronization packets with all the tuples in its three tables. On receiving the packets, the shim generates an per-tuple event on the analyzer, which the latter can use, for example, to populate its shim's tables. An alternate approach would be for the shim to drive the extraction process, so that it can request just subsets of the tables.

Note that, in both cases, the process is initiated by the analyzer. There is a

specific call that allows the analyzer to request synchronization in either direction. Again, the goal is to permit very simple hardware devices.

During the synchronization process, normal operation continues. This means that, while the device is sending its table's entries, the contents of the tables may change, either because the analyzer requested so, or because there were evictions in the device.

4.5.3 Shunt Device

Figure 4.7 shows the device structure. The device captures and injects traffic from two network taps. It has its a copy of the three tables, which it uses (alongside the static filters) to decide whether each packet should be forwarded to the opposite interface, dropped, or shunted to the shim for further processing. The device may also receive packets from the shim, which are injected in the tap opposite to the one from which they were originally captured.

Device States

The shunt device operates in one of two different states. Figure 4.8 shows the device state transition diagram. The normal operation just described occurs when the device is in "working state". The operation in "fail-safe" mode is described below.

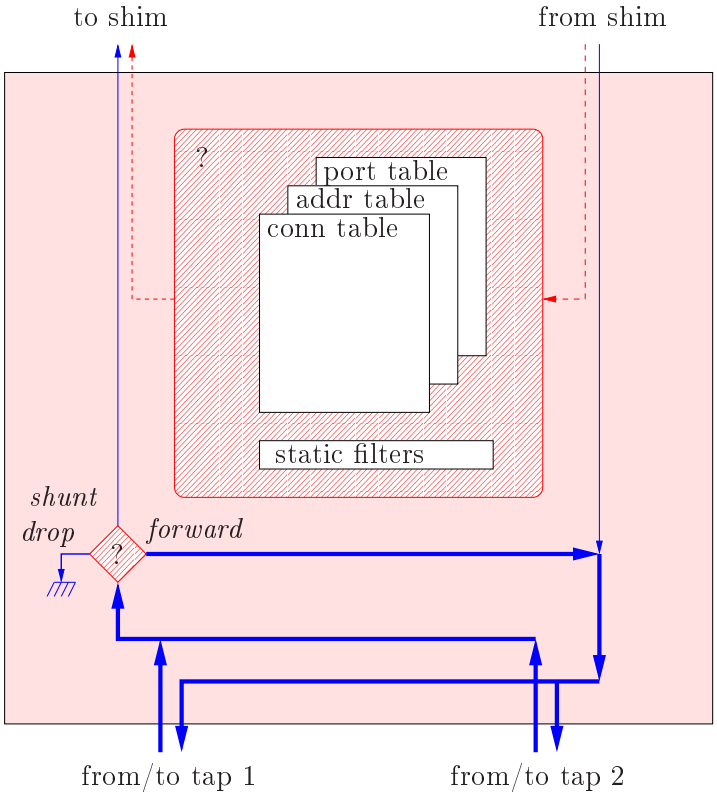


Figure 4.7: Shunt Device Structure

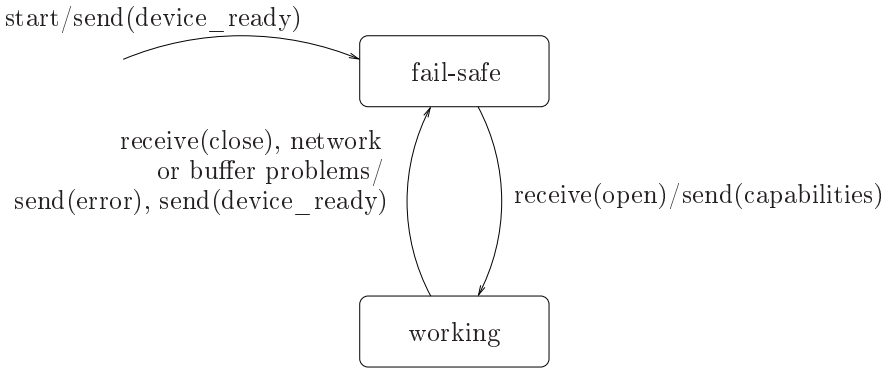


Figure 4.8: Shunt Device State Transition Diagram

Device Filtering

Figure 4.9 shows the full device filtering algorithm in working state, including sampling.

```
1   get a packet

2   query 3 tables and 3 static filters. Choose the entry with highest
    priority, or "default" (shunt) if none matches
        decision = F/D/S (forward,drop,shunt)

3   decide whether the packet must be sampled, according to five actions
    3.1 connection table
            if there is an entry for the packet connection, and the corresponding
            tuple sampling ratio (TSR) is non-zero, use it to decide
            whether to sample the packet or not
    3.2 same for the address table
    3.3 same for the port table
    3.4 3 static filters
            if the filter sampling ratio (FSR) for any of the 3
            static filters is non-zero, use it to decide whether to sample
            the packet or not
    3.5 global sampling ratio GSR
            if the GSR is non-zero, use it to decide whether to sample the
            packet or not

4   decide what to do with the packet
        decision = decision from step 2
        if (steps 3.1 to 3.5 cause the packet to be sampled)
            decision = shunt
            mark the packet as "sampled by action(s)", where
            action is each action that caused the packet to be sampled
```

Figure 4.9: Shunt Device Filtering Algorithm

Layer-2 Transparency

The shunt device is transparent at layer 2. I.e., packets forwarded keep their original Ethernet addresses. The reason is twofold: First, the goal of the device is to be unnoticed by all endhosts, except attackers, who should see it as a very lossy channel. Second, we want to be able to plug the full system in the middle of a link without the need to take care of any link-layer issue.

This decision means that, for data packets going the shunt path, all layers above (and including) layer 2 must be preserved intact. On the other hand, we mentioned in Section 4.5.2 that data packets must include some per-packet information.

Considering that the device-shim communication is carried out using a point-to-point Ethernet connection, there are two alternatives to include the information:

- The generic solution is to encapsulate the data packets into ethernet/IP/UDP packets. This is flexible, but it may cause problems when the shunted packet is maximum-size itself. In this case, transmission of the packet must be done using very large (jumbo) frames, which may present transmission problems.

- For the preliminary implementation used in this project, we have taken advantage of the fact that the device-shim connection is point-to-point, and that, for NIDS purposes, only a handful of network protocols are of interest (namely IP, ARP, and reverse ARP, which can be specified using only 2 bits). We have remapped the original 16-bit ethertype field, using only the first 2 bits, and packed the

per-packet information in the remaining 14 bits. Section B.3 in Appendix B describes the remapping.

Fail-Safe Operation

The goal of the fail-safe state is to permit the device to keep operating safely even when the engine or any of its parts has crashed. Inline packet-processing adds stringent requirements to the system reliability: In case of problems in the shunting system, some network operators will prefer to fail-open, meaning to disable the shunting system and forward all the traffic. Network operators with different requirements may prefer to cut the connectivity of their site before letting traffic pass without having been passed through the analyzer.

If the shim sends a Close message to the device, or the device detects it is malfunctioning, the latter will go into fail-safe mode. In the second case, a Close message is sent back to the shim, if possible, accompanied by an Error message, if needed.

In the device's fail-safe state, all data packets received are either directly forwarded to the other network tap (fail-open mode), or dropped (fail-close mode). The fail-over mode can be set by the analyzer through the shim API.

Device Implementation

Current network interface cards do not provide enough functionality to classify traffic as the Shunting architecture requires. Therefore, we are working on the development of a specific piece of hardware that will provide it. Nick Weaver from the International Computer Science Institute is building a hardware device that will work as the shunt device.

While we are building the hardware device, we have written a software simulator of the device. This "software device" implements the full functionality of the hardware device. It has been written using Click [Morris et al., 1999].

The software device also provides a cheap Shunting system for low-bandwidth links (100 Mbps or less), and an easy-to-debug testbed.

4.5.4 Analyzer-to-Shim API

Table 4.4 and Table 4.5 describe the API exported by the shim to the analyzer. It includes functions (messages which the analyzer uses to program the shim) and events (calls initiated by the shim to report something to the analyzer). As our first implementation uses Bro, functions and events are described using the Bro script language.

Note that most of the API functions have a one-to-one correspondence in the shim-to-device API. The only additions are shunt_inject_packet and shunt_drop_packet, that permit explicit forwarding or dropping of a packet at the shim. The exact

semantics of each API function and event are described in Section B.1 in Appendix B.

4.5.5 Shunt Shim

Figure 4.10 shows the shim structure. The shim receives a packet from the device, and has a copy of the tables to carry out filtering. If it decides to shunt a packet, it sends the packet to the analyzer, which processes it. After the processing, and if the analyzer injects back the packet, it is filtered again in the shim, and if finally accepted, sent back to the device for injection into the wire.

Note that the shim structure is very similar to that of the device (compare with Figure 4.7). The device receives traffic from two network taps, and filters it using its tables. The tables are likely very limited, as the device may be implemented in hardware with limited resources, and are controlled by SHIP traffic from the shim. Most of the traffic should be directly forwarded, and therefore injected in the tap distinct to the one where the packet was received. If the decision is to shunt, the traffic is pushed up to the shim.

The shim receives traffic from the device, and filters it using its own tables. The shim tables are unlimited, as the shim typically runs on an off-the-shelf host, and are controlled by the analyzer through the shunt API. Most of the traffic should be sent to the analyzer (otherwise the shim is performing badly) and processed there. If reinjected by the analyzer, the packet is filtered again (to take into consideration changes in the tables caused by the processing of the very same packet), and then

Table 4.4: Bro Shunt Access API (Functions)

function	**parameters**	**explanation**
shunt_open		open the device
shunt_close		close the device
shunt_reset		reset the device
	hard	whether the reset must be hard or soft
shunt_inject_packet		inject the packet currently being analyzed
shunt_drop_packet		drop the packet currently being analyzed
shunt_associate_conn		insert a tuple in the connection table
	conn_id	connection ID
	forth_action	whether packets in the forth direction must be forwarded/dropped/shunted
	back_action	whether packets in the back direction must be forwarded/dropped/shunted
	priority	tuple priority
	sampling	tuple sampling ratio
shunt_associate_addr		insert a tuple in the address table
	address	the address
	forth_action	whether packets in the forth direction must be forwarded/dropped/shunted
	back_action	whether packets in the back direction must be forwarded/dropped/shunted
	priority	tuple priority
	sampling	tuple sampling ratio

Continued on next page

function	parameters	explanation
shunt_associate_port		insert a tuple in the port table
	port	the port
	forth_action	whether packets in the forth direction must be forwarded/dropped/shunted
	back_action	whether packets in the back direction must be forwarded/dropped/shunted
	priority	tuple priority
	sampling	tuple sampling ratio
shunt_deassociate_conn		evict a tuple from the connection table
	conn_id	connection ID
shunt_deassociate_addr		evict a tuple from the address table
	address	the address
shunt_deassociate_port		evict a tuple from the port table
	port	the port
shunt_get_status		request device status
shunt_get_statistics		request device statistics

Table 4.5: Bro Shunt Access API (Events)

event	**parameters**	**explanation**
shunt_associate_conn_event		a tuple was inserted into the connection table
	conn_id	connection ID
shunt_associate_addr_event		a tuple was inserted into the address table
	address	the address
shunt_associate_port_event		a tuple was inserted into the port table
	port	the port
shunt_deassociate_conn_event		a tuple was evicted from the connection table
	conn_id	connection ID
shunt_deassociate_addr_event		a tuple was evicted from the address table
	address	the address
shunt_deassociate_port_event		a tuple was evicted from the port table
	port	the port
shunt_query_decision_event		the shim took a packet decision
	pkt_hdr	packet header
	decision	the decision taken
	reason	rationale behind the decision
shunt_status_event		device status received
shunt_statistics_event		device statistics received

sent back to the device for injection in the wire.

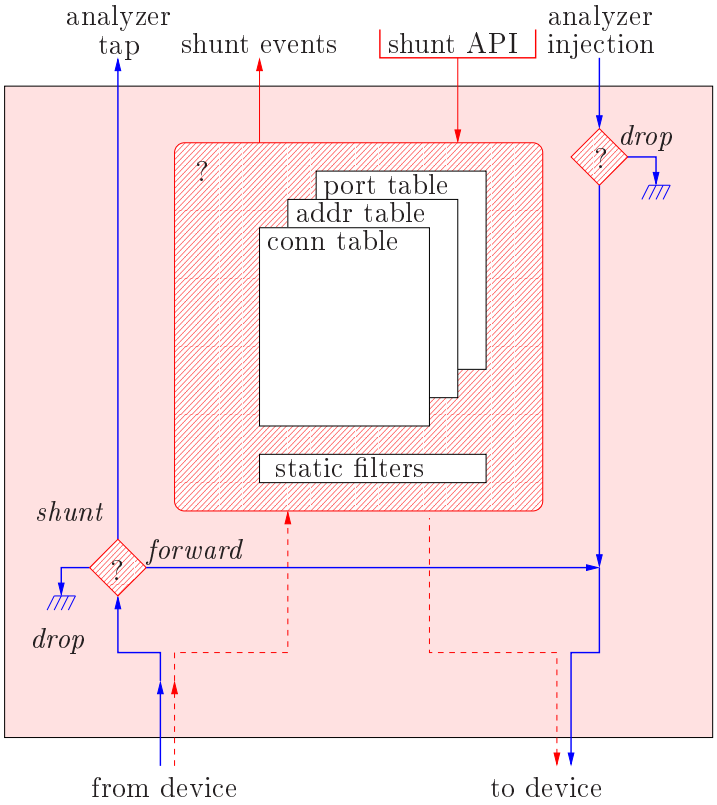


Figure 4.10: Shunt Shim Structure

4.5.6 Analyzer

We use Bro with some extensions as back-end software NIPS driving the engine. The list of modifications carried out in Bro include (a) reading packets from the shim, instead of the network tap, (b) injecting processed packets back into the shim, (c) extending Bro analyzers to control the shunting system, and (d) having direct read access to the connection, address, and port tables by exporting them into Bro. The last modification permits access to the tables from Bro scripts (though read-only),

and also making the tables persistent.

Note that the analyzer extensions are configurable. We describe the ones set by default. Those include so far the following items:

- Bro portscanning analyzer (scan.bro) includes a function that is typically called when the analyzer detects a portscan (drop_address()). We have extended such function to include an entry in the address table with a *drop* yield, so that further packets from the mentioned address are dropped directly in the shunt device.

- Bro's connection analyzer (Conn.cc) includes SetSkip(), a function that analyzers call when they do not need to see more payloads of a given connection. It is used, for example, by the SSH, SSL, and login analyzers. We have extended the function to insert a tuple in the connection table, with a *forward* yield.

- Any TCP or UDP connection whose protocol is not supported by any Bro analyzer is also inserted in the connection table, with a *forward* yield. The rationale is that, if no application-layer analyzer is available, then the only events that need be reported are those related to the network and transport layers, which are indeed captured by the *shunt* static filter.

- Multicast traffic causes the corresponding multicast address to be included in the address table with a *forward* yield.

- The SSH analyzer has been extended to set a *forward* yield in the connection table when it thinks the connection is OK.

- Bro's FTP analyzer (ftp.bro) has been configured to set a *drop* entry for FTP data and control connections where buffer overflow attacks based on excessive filenames are detected; and a *forward* entry for all other FTP data connections. Both passive (PASV and EPSV commands) and active FTP data connections (PORT and EPRT commands) are identified from the FTP control connections. In both cases, if the FTP analyzer does not see anything strange in the FTP control connection, an entry corresponding to the data connections is set to *forward* in the device connection table

- The HTTP analyzer has been extended as follows: If the configuration includes the HTTP request analyzer, all the requests are shunted for in-Bro analysis. If the configuration includes the HTTP reply analyzer, all the replies are shunted for in-Bro analysis. Otherwise the replies are directly forwarded (even while the requests may still be shunted).

  If an HTTP connection's request is in Bro's list of dangerous (sensitive) URLs, or the contents of the reply are deemed dangerous, a *drop* entry is set in the device's connection table.

- When a TCP connection is finished, Bro's TCP connection analyzer (TCP.cc) raises a connection_finished event. We have extended this event to remove the

corresponding entry from the connection table.

4.6 Evaluation

4.6.1 Project Status

We have written the shim and extended a well-known NIDS (Bro) to be used as the engine's analyzer. Note that this transforms the NIDS to a *de facto* NIPS, allowing Bro to instantly block attack traffic.

We are still working on the implementation of the hardware device. In order to be able to analyze the shunt performance, we have written a "software device", a program that simulates the functionality of the hardware shunt device, including the use of a fixed amount of resources. The software device permits us (a) gaining understanding of the filtering effect of the shunt device, (b) obtaining a hint on the net performance benefits, and (c) allowing the running of the Shunt architecture by software-only means in low-speed links (sort of a poorman's shunt architecture).

We have so far extended Bro's SSH, HTTP, FTP, and generic TCP connection analyzer to take advantage of the shunt architecture. We are working in extending more Bro analyzers, to ensure full use of the shunt capabilities.

The code consists of 16 K lines, mainly heavily-commented C++ code, from which 6.4 K are used in common code (SHIP protocol and table implementation), 4.6 K are used for the shunt shim, 2 K for the analyzer extension (including some Bro scripts),

and 3 K lines for the shunt simulator (including some click glue).

We have run several experiments to measure the benefit obtained by the shunting architecture. These experiments cover four different aspects:

The main benefit of the shunt architecture is that the NIPS has to process only a fraction of the total traffic being analyzed. Shunting filters the traffic that the analyzer receives, and most of the traffic is directly processed in the shunt device. Section 4.6.3 quantifies this benefit, measured as the *filtering ratio*, which is defined as the amount of traffic that a NIPS has to process when used as the analyzer in the shunt engine, compared to what the same NIPS would have to process if running without shunting.

Second, this shunt filtering effect has as natural consequence an enhancement in the system performance: If the NIPS running as the analyzer in the shunt engine has to process less traffic than when running without shunting, it should also run faster. While it is hard to evaluate this benefit without a hardware device, Section 4.6.4 tries to provide some hints.

The third effect we have investigated is the influence of limited tables in the shunt device filtering performance. We want to know how much we should augment the shunt device capabilities in order to deal with a real traffic environment. Section 4.6.5 studies the filtering ratio for several table configurations.

Finally, Section 4.6.6 describes the experience obtained from running the shunt architecture in a live environment.

4.6.2 Trace Description

This Section describes the different traces in which we have run Bro with shunting.

Isolated Experiments

First, and in order to double check that the shunt is working fine, we have used 2 isolated port traces, namely *ssh-1* and *www-1*.

ssh-1 is a 45 min trace taken at Lawrence Berkeley National Lab (LBL) DMZ on November 2004. It consists of port 22-traffic only, and accounts for 757 connections, 2 M packets, or 1 GB (an average of 530 bytes/packet).

www-1 is a 25 min trace taken at LBL DMZ in September 2004. It consists of port 80 traffic from or to the LBL web server. The trace accounts for 2320 connections, 150 K packets, or 100 MB (an average of 670 bytes/packet).

tcp-1 Trace Description

A more realistic trace (*tcp-1*) was obtained at the LBL DMZ, whose link is 1 Gbps. It consists of TCP-traffic only, and accounts for 1.2 M connections, 127 M packets, and 113 GB (an average of 892 bytes/packet). The trace was taken during working hours on a weekday, in September 2005. Its total duration is 2 hours (an average bitrate of 126 Mbps).

4.6.3 Shunt Filtering Ratio

The most important benefit of the shunt architecture is the reduction in the amount of traffic the analyzer must process. We define the *filtering ratio* as the proportion of the analyzed traffic that a NIPS has to really process when used as the analyzer in the shunt engine. For example, a filtering ratio of 10% means that, from all the traffic in the wire, the analyzer really processes 10%, and the remaining 90% is processed in the shunt device.

In order to measure the filtering ratio, our first experiment uses an unlimited-size shunt device simulator. Bro was configured to use the shunt capabilities only for the three Bro analyzers that have been modified to do so (HTTP, SSH, and FTP). In order to permit full HTTP payload inspection, Bro's HTTP analyzer was configured so that it requests shunting for all traffic in both the server-to-client (the packets that conform the bulk data transmission) and client-to-server direction (the ACKs that clock the bulk data transmission).

Isolated Experiments

We ran some introductory experiments in the *ssh-1* and *www-1* traces. In the first case, the filtering ratio is extremely small, around 0.2% of the packets and 0.05% of the bytes.

In the *www-1* case, the filtering ratio is 100%: There are no savings at all, as the shunt is instructed to divert all HTTP traffic to the analyzer for processing.

tcp-1 Results

From the total amount of traffic, all of the 1.2 M connections had at least one packet shunted. The traffic shunted accounted for 30 M packets and 20 GB, i.e., a filtering ratio of 24% of the packets or 18% of the bytes.

In order to understand what is being shunted by the device, Table 4.6 shows the decomposition of the shunted traffic.

category		percentage on shunted traffic	
		packets	bytes
1	flags	13.7	1.32
2	AUS	71.6	85.7
2.1	FTP	0.05	0.01
2.2	SSH	0.02	0.00
2.3	HTTP	71.6	85.7
2.3.1	src HTTP	43.6	80.1
2.3.2	dst HTTP	28.0	5.61
2.3.2.1	HTTP ACKs	24.0	1.57
2.3.2.2	HTTP GET/POST	4.03	4.04
3	AAS	14.3	12.99
3.1	port 25	3.47	4.21
3.2	port 8000	1.36	0.49
3.3	port 443	5.55	4.54
3.4	port 993	2.62	2.44
3.5	port 995	0.99	1.03
4	NAN	0.33	0.03

Table 4.6: Shunted Traffic Decomposition, *tcp-1*

The description of the different categories is as follows:

1 (flags) corresponds to TCP flags (TCP segments with the SYN, FIN, or RST flag set). This traffic is required in order to monitor transport protocols, and to account for connections.

2 (AUS) corresponds to traffic for which Bro has an analyzer, and this analyzer has been modified to take advantage of the shunt. Currently this includes the HTTP, SSH, and FTP analyzers.

2.3.1 corresponds to traffic with port 80 as *source*. This HTTP traffic corresponds to the bulk data being transmitted in HTTP connections. Note that this accounts for close to 85% of the bytes.

2.3.2 corresponds to traffic with port 80 as *destination*. This is HTTP traffic, and can be further subdivided as:

2.3.2.1 corresponds to empty ACKs that the client users to clock data transmissions from the server.

Note that the typical empty ACK is a small packet (43 bytes in average), and therefore any effort to reduce this traffic category will be only important on packets, not in bytes.

2.3.2.2 corresponds to the HTTP GET/POST lines that the client uses to request data transmissions from the server.

3 (AAS) corresponds to ports for which Bro has an analyzer, but it has *not* been modified to use the shunt. For these ports, all packets are shunted, so that the corresponding Bro analyzer can process them.

The main source of AAS traffic is HTTPS (port 443), which accounts for 4.5% of the shunted bytes, and SMTP (port 25), which accounts for 4.2% of the shunted

bytes. The HTTPS and SMTP Bro analyzers are the two main candidates to be instructed to use shunting.

4 (NAN) corresponds to ports for which Bro has no analyzer. Connections whose destination port has no analyzer in Bro are shunted as soon as Bro instantiates the corresponding per-connection data.

Conclusions

The total bitrate that the analyzer receives gets reduced to less than one fifth of the original bitstream.

Moreover, from this 20% of the total bytes that are shunted, close to 85% of the bytes are caused by HTTP traffic, including 80% by HTTP payloads. Section 4.7.1 proposes a more fine-grain mechanism to deal with HTTP and other similar protocols.

4.6.4 Shunt Performance

The filtering ratio provides an idea of the applicability of the approach for different link bandwidths. On the other hand, this ratio does not necessarily translate into a proportional reduction in the amount of resources used in the analyzer.

For once, it seems clear that as the filtering ratio diminishes and less traffic reaches the analyzer, this filtered stream has been selected for analysis, and therefore is more likely of interest for the Intrusion Detection Analyzer. We expect the amount of analysis required per packet to be larger in the shunted stream than in the original

one, and the savings in performance to be smaller than the savings in the shunted bitstream.

Also, it may be the case that the main factor in the NIDS performance is not the number of processed packets, but the number of processed connections, which the static *shunt* filter at the device ensures that are always seen by the analyzer.

In order to check the savings in resources, we studied the performance of different configurations of Bro and shunt processing the three experiment traces, measuring the running (user plus system) time of each of the configurations. The exact configurations were as follows:

T1 plain Bro (no shunt) on the original experiment trace, with the following analyzers loaded: scan, ssh, ftp, http, http-event, http-request, http-reply, notice, conn, and weird.

T4 Bro using shunting on the original experiment trace, without the simulator device, and using the same set of Bro analyzers.

T5 Bro using shunting on the full trace, with the simulator device (*simdev*), and using the same set of Bro analyzers.

T2 Bro using shunting on the "shunted trace", a reduced trace composed of the traffic that was shunted when running T5 on the original trace. T2 should be approximately equivalent to running the shunt engine behind a real hardware shunt device, which shunts to the engine the contents of the shunted trace.

Table 4.7 shows the results of running the four configurations in the three different traces.

trace	configuration	time (seconds)		
		total	*user*	*system*
ssh-1	T1 (plain Bro)	5.4	4.2	1.2
ssh-1	T4 (Bro+shunt)	12	11	1.0
ssh-1	T5 (Bro+shunt+simdev)	20	19	1.0
ssh-1	T2 (Bro+shunt on shunted trace)	0.25	0.24	0.01
www-1	T1 (plain Bro)	6.5	6.4	0.18
www-1	T4 (Bro+shunt)	7.3	7.2	0.18
www-1	T5 (Bro+shunt+simdev)	8.6	8.4	0.18
www-1	T2 (Bro+shunt on shunted trace)	7.3	7.1	0.16
tcp-1	T1 (plain Bro)	2750	2500	250
tcp-1	T4 (Bro+shunt)	3050	2800	250
tcp-1	T5 (Bro+shunt+simdev)	3450	3200	250
tcp-1	T2 (Bro+shunt on shunted trace)	2550	2500	50

Table 4.7: Shunt Performance Results

The benefit in the SSH trace case is impressive for the T2 case. Bro with shunting runs 22 times faster than the Bro without shunting.

The poor performance of the case T4, which is two times slower than running Bro by itself, states that filtering traffic in the shim is slow, as compared to capturing the packet and just discarding it. From visual inspection of a profile of the running, the main causes of the slowdown of the shim are (a) the static filter processing in the shim (17% of the time is spent there), (b) the management of local statistics in the shim (15%), and (c) the cost of using Bro tables so that they are accessible from Bro scripts (15%). We believe we can reduce these three costs by (a) considering that the static filter processing will be done correctly at the device, and therefore does not

need being repeated at the shim; (b) optimizing the performance, or disabling local statistics; and (c) exporting the table entries to the analyzer as functions, instead of tables, which would allow for a faster implementation.

The poor performance of the case T5, which is four times slower than running Bro by itself, states that the shunt device implementation is also slow. We also profiled this case, and realize that 75% of the processing time is caused by the table implementation in the device, which focuses on saving space by storing each entry's key, yield, and valid bit in the minimum number of possible bits. This has demonstrated to be a poor implementation decision, which we are currently fixing.

In the *www-1* trace, T1 runs 12% faster than T2. This extra cost of T2 (which analyzes the same traffic, as all HTTP traffic is shunted to the engine) can be explained by the filtering of the traffic in the shim to check that the device did its work well.

In the *tcp-1* trace, T2 ran 10% faster than T1. While this number is not impressive, the system time consumed by T2 was one fifth of the system time consumed by T1. This makes sense as the size of the shunted trace in the *tcp-1* case is approximately one fifth of the original trace.

We think there is a lot of leeway in the user processing to make T2 run much faster than in T1. Also, the BTL approach described in Section 4.7.1 should be able to help with a reduction in the amount of HTTP traffic that reaches the engine.

We also want to move the shunt device to the kernel, adding is as an in-kernel

filtering alongside the BPF filtering. This is relatively safe, as the size of the tables is limited and well-known beforehand, and the operations available are relatively simple (add or delete entries in the tables). This should provide added savings as the number of packets requiring a context exchange will be much smaller.

4.6.5 Device Limited-Size Influence

This Section studies the effect of the limited size of the shunt device in the amount of traffic that reaches the analyzer in the engine.

A shunt device is a hardware device, and therefore the resources it can use are limited. We assume the hardware device used to implement it would have a few MB of fast memory (SRAM), and some more slower memory (DRAM).

The tables have been implemented in the shunt device simulator as cache tables with a pseudo-random (cryptographically secure) hash function (H3 [Carter and Wegman, 1979]).

From the three tables used in the shunting architecture, we expect the connection table to be the most useful in filtering traffic, and at the same time the one requiring more space and the most used one. The port table is very minor (an exhaustive table covering all the possible ports with the mentioned 10 bits/entry requires 80 KB), and [Weaver et al., 2004] shows how to track all external IP addresses of note for a fairly large (several thousand hosts) site with a 4 MB table.

We therefore focus our efforts on investigating the effect of the limited size of the

connection table.

We have considered a connection table with 1 MB, 2 MB, and 4 MB of space, which at 16 bytes per entry, fits 64 K, 128 K, and 256 K entries, respectively. We have also considered three different levels of associativity, namely 2-way, 4-way, and 8-way associativity. The three values are small enough as to permit quick read-only packet processing. The last parameter whose effect we have studied is the eviction policy in the cache table. We have considered random and Least Recently Used (LRU) eviction.

For any combination of the three parameters, the filtering ratio result is compared with the filtering ratio assuming an infinite table. What we want to know is how much traffic will reach the engine for different cache configurations.

For all the experiments, we have used the *tcp-1* trace described before. The trace is composed of 1.2 M connections, and the analyzer has a maximum instant demand (the maximum number of entries in the table connection that are needed at any given time) of 200 K entries. Note that this number is close to 3 times larger than the capacity of the smallest size configuration.

Figure 4.11 shows the number of entries requested by the analyzer (top line), and the number of entries used for different combinations of capacity and associativity. Results using LRU eviction are very similar to those using random eviction, and therefore are not shown.

We can see that, in the configurations with 64 K and 128 K entries, the analyzer

reaches the maximum capacity of the table before the end of the experiment. While it can be argued that the 256 K entry configuration would eventually run into capacity problems, and therefore its long-term performance cannot be understood, this is not the case for the 64 K and 128 K entry ones, which do work in a stable state for a large part of the experiment.

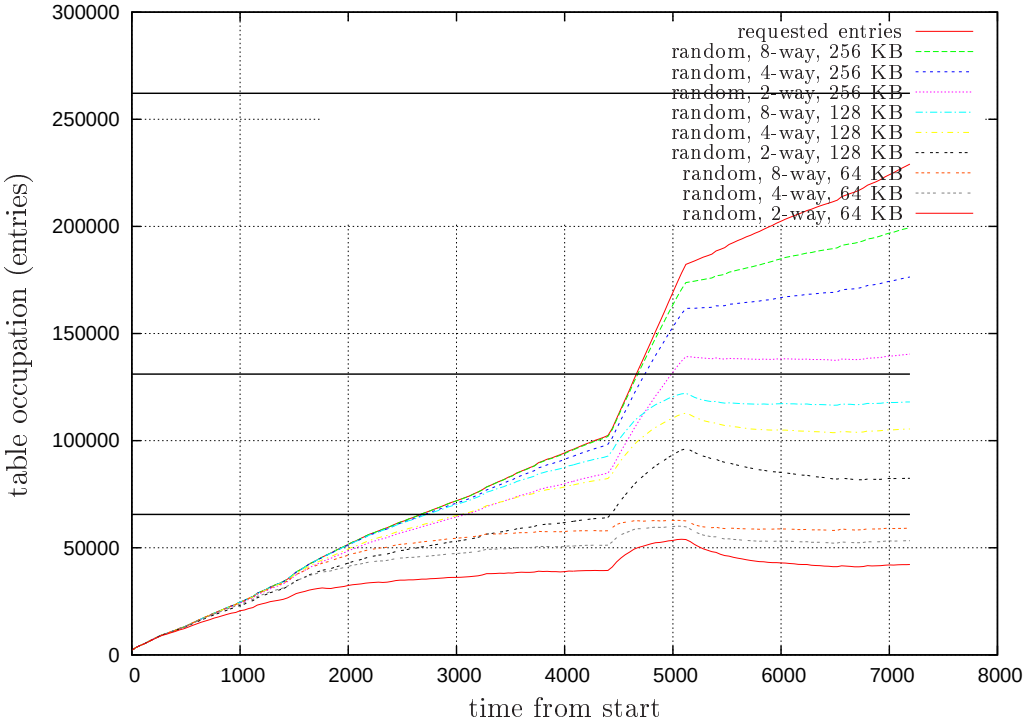


Figure 4.11: Table Size Occupation

Figure 4.12 shows, from the total bytes shunted during the experiment, the ratio of these bytes that would have not been shunted if the device had infinite capacity. This measures how much extra traffic is a limited-size device sending to the shunt, or in other words, which is the penalty of a limited-size device.

We can see that, in general, LRU causes around one order of magnitude less extra

incorrectly shunted traffic than random eviction. Moreover, LRU gets more benefit from higher associativity than random eviction. This is also expected, as LRU gets can make better use of a higher associativity to optimally evict the tuples.

The drawback of LRU is that every table read operation requires a write operation, in order to account for the least recently used entry.

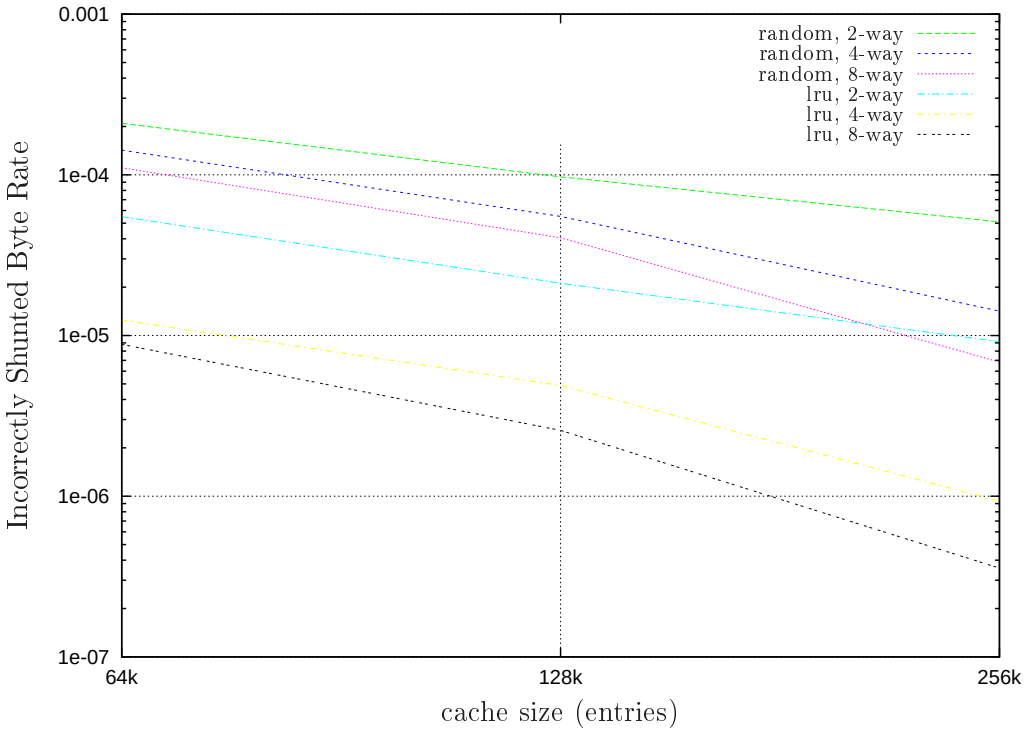


Figure 4.12: Incorrectly Shunted Byte Rate

Figure 4.13 shows, from the total bytes shunted during a given time period (160 sec), the percentage of these bytes that would have not been shunted if the device had infinite capacity. This measures how much extra traffic is a limited-size device sending to the shunt, in other words, which is the penalty of a limited-size device.

The Figure only presents results for the 64 K entry table cases (the cases with more entries have lower percentages of incorrectly shunted bytes). The results show that higher associativities and LRU eviction perform consistently better than lower associativities and random eviction, respectively. This was expected as reissuing entries permits LRU eviction and a larger associativity to optimize the composition of the table by trial and error.

The most interesting aspect of the experiments, though, is how small the effect of limited-size devices is. In the worst case (random eviction, 2-way associativity, 64 K entries), the maximum amount of incorrectly shunted data is a meager 0.07% of the total traffic. This is also true when using smaller time periods: The maximum amount of incorrectly shunted data in a 10 sec period is 0.7% of the total traffic. (The corresponding number for bytes are 0.2% for the 160 sec period, and 0.4% for the 10 sec period).

The main reason why the percentages are so small is table reissuing. When the shim (which has infinite-size tables) detects that a packet shunted by the device should have been processed directly by the device, it reissues the table entry to the device. This means that at most one packet would be dealt with incorrectly at the device: After it reaches the shim, the latter reinstates its connection table entry.

By reissuing entries after each miss, the shim quickly finds the active connections. Packet connections present a very strong locality of reference, both temporal and spatial: Packets from the same connection are typically close in time, and there

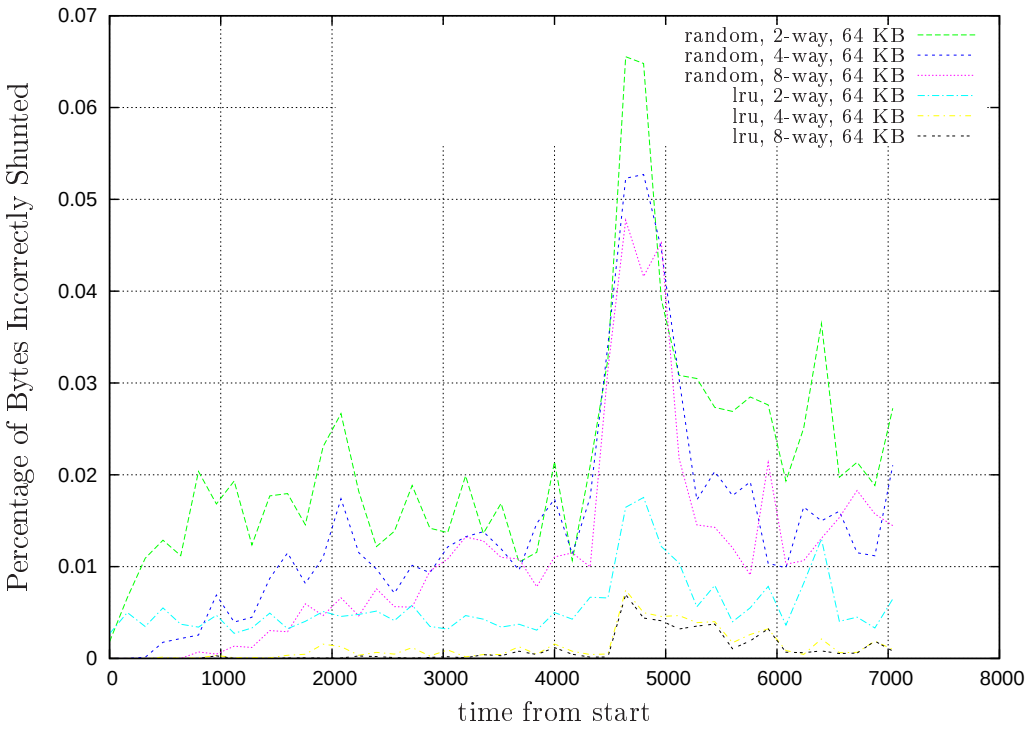


Figure 4.13: Incorrectly Shunted Bytes

are not too many active connections present at a given time, compared to the total number of connections ever seen in the wire.

There could be a problem if, in an N-way associative cache, $N + 1$ connections hashing to the same cache position were active at the same time (thrashing). Otherwise, either LRU or random will quickly evict the non-active connection (though LRU faster).

Comparing the extra amount of shunted data because of the limited-capacity device with the results in Section 4.6.3, the main conclusions are that LRU is better than random (with higher associativity, up to one order of magnitude less extra traffic), but that the percentage of extra shunted traffic with a sensible-size table

(1 MB) is so small, that the extra in-device accounting needed in order to implement LRU is not worth.

4.6.6 Live Shunting

We have also run Bro with the shunt in a real, low-bandwidth environment, for over 262 hours (11 days). The total traffic processed was 34 M packets, or 9.8 GB (36 pkts/sec and 83 Kbps, or around 300 bytes/packet).

The host used for shunting is a 2-processor, Pentium (Xeon) at 3 GHz CPU (hyperthreading enabled) with 2 GB of memory running Linux (2.6.9 kernel). This host was used to route traffic between the internet and four hosts, including some researcher's desktops and laptops.

Bro was configured to use the software shunt device, and the shunt capabilities only for the three Bro analyzers that have been modified to do so (HTTP, SSH, and FTP). The HTTP analyzer was configured to request forwarding of all traffic in the server to client address, so that HTTP entity bodies (but not the HTTP entity headers) are either forwarded or dropped at the simulator device, but never shunted. Note that this approach is not the same than the one described in Section 4.7.1, where all HTTP entity bodies are shunted, independently of their type.

From the full traffic processed at the shunt device, 8.9% of the packets and 3.5% of the bytes were shunted. This corresponds to the device having shunted to the analyzer an average 3.2 pkts/sec, 2.9 Kbps, and 115 bytes/packet. Note that the

average shunted packet is less than half the size of the average packet seen at the device, which makes sense due to the large percentage of empty TCP packets in the shunted traffic.

The peak amount of traffic shunted to the analyzer in a one-second interval was 750 pkts/sec, or 800 Kbps. During this peak, the time used by Bro to analyze the packets accounted for 16% of the wallclock time. The peak happened around one week after the experiment was started, so it was not due to cold start. No packet losses were reported by any of the pcap devices used to capture traffic.

Table 4.8 shows the decomposition of the 3 M packets (345 MB) shunted.

type		percentage on shunted traffic	
		packets	bytes
1	flags	26.54	13.62
2	AUS	51.18	48.74
2.1	FTP	0.00	0.00
2.2	SSH	4.68	2.45
2.3	HTTP	46.50	46.30
2.3.2	dst HTTP	45.03	45.74
2.3.2.1	HTTP ACKs	40.94	16.51
2.3.2.2	HTTP GET/POST	4.10	29.23
3	AAS	14.68	30.52
3.3	port 443	4.53	11.31
3.6	port 139	4.65	14.20
3.7	port 111	2.51	1.50
3.8	port 53	2.99	3.50
4	NAN	7.61	7.12

Table 4.8: Shunted Traffic Decomposition, Live Traffic

One third of the shunted traffic is composed of the HTTP entity headers, and one fourth is composed of traffic corresponding to ports whose corresponding Bro

analyzer has not yet been modified to take advantage of the shunting architecture.

4.7 Future Work

4.7.1 Expiring Entries (BTL)

Justification

The normal operation for the shunt architecture is to shunt the first few packets of a connection, so the analyzer can take a decision on whether it wants to keep receiving packets from it or no. Once the analyzer takes a decision, we use it to forward/drop (or shunt) any further connection traffic.

This operation mode works if the life of a connection consists of some application-layer metadata and control exchange, followed by a bulk data transfer. This is the case for SSH, FTP, and non-persistent connections in HTTP. But, what happens when metadata and data are interleaved in a connection?

A case example of this behavior is persistent connections in HTTP 1.1 [Fielding et al., 1999]. Persistent HTTP connections reuse the same TCP connection to serve multiple requests, therefore avoiding the overhead of opening a TCP connection for each request.

Persistent HTTP connections present several advantages: (a) They reduce the total amount of traffic in the wire, as the TCP connection establishment and teardown handshakes are needed only once per connection; (b) they reduce the latency of

loading all objects but the first one, as (again) the TCP connection establishment handshake is only needed once; and (c) they increase the performance of small request transfers, as they are carried out in a TCP connection outside of slow start [Jacobson and Karels, 1988].

Figure 4.14 shows an example of the structure of an HTTP persistent connection. In the left column, C represents the HTTP client, and S the HTTP server. The right column shows the contents of the requests and replies.

HTTP Server to Client replies consist typically of one or several entity transfers [Fielding et al., 1999]. Each entity transfer is composed of an entity body (the data that is being served by the reply), preceded by an entity header describing it. The latter is composed of a series of one-line fields, including among others the object type, its encoding, and its length.

Let's consider the situation where a network operator running Bro wants to analyze the entity headers in HTTP responses, but not some of the associated entity bodies (for example, large binaries). If a connection is not persistent, the analyzer can be instructed to check the entity headers, and if the body being described is not of interest, or there is no analyzer available for the media type, instruct the device to *forward* all further traffic associated to the connection.

With persistent connections, though, the analyzer cannot assume that the connection used at one moment for sending an uninteresting entity body will not be used later for an interesting one. Current shunt capabilities (connection, address, and port table,

direction	*contents*
$C \leftrightarrow S$	[TCP Handshake]
$C \rightarrow S$	GET page.html HTTP/1.1\r\n
	Connection: KeepAlive\r\n
	Keep-Alive: $\cdots$
$S \rightarrow C$	HTTP/1.1 OK\r\n
	$\cdots$header$\cdots$\r\n
	\r\n
	$\cdots$data$\cdots$
$S \rightarrow C$	$\cdots$data for first ADU$\cdots$
$C \rightarrow S$	GET image1 HTTP/1.1\r\n
	Connection: KeepAlive\r\n
	Keep-Alive: $\cdots$
$S \rightarrow C$	HTTP/1.1 OK\r\n
	$\cdots$header$\cdots$\r\n
	\r\n
	$\cdots$data$\cdots$
$S \rightarrow C$	$\cdots$data for second ADU$\cdots$
$C \rightarrow S$	GET image2 HTTP/1.1\r\n
	Connection: KeepAlive\r\n
	Keep-Alive: $\cdots$
$S \rightarrow C$	HTTP/1.1 OK\r\n
	$\cdots$header$\cdots$\r\n
	\r\n
	$\cdots$data$\cdots$
$S \rightarrow C$	$\cdots$data for third ADU$\cdots$

Figure 4.14: HTTP Persistent Connection Example

plus static filters) do not permit forwarding the entity bodies while at the same time shunting the entity headers: Both share the same connection, and packets from the same connection are always dealt with in the same manner. All packets must go through the analyzer, and the shunt benefit for that connection gets reduced to zero.

An added problem is that, in HTTP/1.1, unless either the client or the server state explicitly that they want non-persistent connections, connections are persistent (Section 8.1.2 in [Fielding et al., 1999]). The analyzer only knows that it should not expect more entities in a connection when it sees the FIN segment from the client that effectively closes the TCP connection. It must therefore dealt with all HTTP/1.1 connections as if they were persistent.

This means that, in order for the analyzer to access to the HTTP responses, the device must shunt to the analyzer all the packets corresponding to default connections, even if they end up being used as non-persistent.

BTL Description

The solution we propose to address HTTP persistent connections is to extend the connection table yield with an application-layer byte counter, which we call *BTL*, or bytes-to-live. The basic idea consists of the connection table still producing an action with a priority, but before doing so, the length of the matching packet's application-layer contents is subtracted from the value of the entry's BTL counter. When the BTL counter reaches zero, the entry is removed automatically from the

table.

Figure 4.15 shows the operation of the connection table under the BTL approach. Times goes from the top to the bottom. The left column depicts the state of the connection table. (The yield shows only the main action and the BTL counter.) The column in the middle represents packets processed from the highlighted connection. The right column represents the action carried out by the device (we assume neither the other two tables nor the static filters match any of the three packets).

The first packet matches the highlighted column. It is forwarded, and the BTL value of the entry is reduced in the amount of application-layer bytes in the packet (1000 bytes in this case). The second packet also matches the highlighted column, and therefore is dealt with the same way. But, when subtracting the total amount of application-layer bytes of the packet, the BTL counter reaches zero, and the entry is automatically removed. The third packet does not match any entry in the connection table, and therefore is shunted.

The BTL counter allows the analyzer to specify expiring entries. For example, if the analyzer sees an HTTP entity header where the Connection−Length header states that the entity body will occupy 100 KB, it can set a *forward* entry associated to the connection, but with a BTL value of 100 KB. The shunt device will keep forwarding packets, and discounting the forwarded application-layer length from the BTL. Once the BTL reaches zero, the entry is removed, and further packets are shunted.

Note that this solution will not only work for HTTP persistent connections, but

Figure 4.15: BTL Operation

also for any other application-layer protocol where the size of the bulk data transfer is obtained from the application-layer header (in HTTP this is possible, unless the "chunked" transfer is used),

The BTL approach has two main drawbacks: First, it requires a per-packet write for traffic belonging to a connection with a non-zero BTL, which may be expensive. Second, if packets get reordered just near the time when the BTL entry is about to expire, then some headers for the next chunk (which the NIDS wanted to see) might get forwarded, and some data from the current chunk (which the NIDS did not want to see) might get shunted. This means the NIDS might miss some important headers, which might cause it to miss attacks. This could happen either by mischance (network re-ordering, duplications, or end-host retransmission) or malice (deliberate evasion).

In order to fight these two drawbacks in TCP traffic, our implementation of BTL resorts to using TCP sequence numbers instead of application-layer lengths. Instead of specifying the maximum amount of traffic that is to be processed according to the entry, the analyzer specifies the last sequence number that will be processed accordingly to the entry. This approach solves both BTL drawbacks for TCP traffic. It is not applicable to UDP, where the application-layer data order is not identified explicitly.

Finally, some scenarios will still justify non-expiring entries, and therefore a fixed value could be used to denote them. In a hardware implementation of a device, it may make sense for efficiency reasons to avoid using a full counter for each non-expiring

entry, and therefore to have two connection tables, one with BTL, and the other without BTL.

Evaluation

This Section analyzes the effect of adding a BTL counter in the processing of the tcp-1 trace discussed previously. We are interested in knowing (a) the distribution of content types in the HTTP traffic, to get an idea of which ones could be dealt with using BTL, and which ones deserve full engine analysis, and (b) the percentage of HTTP traffic that could be forwarded directly when using the BTL approach.

Figure 4.16 shows a breakup of the HTTP contents in *tcp-1*. The HTTP contents include all bytes corresponding to HTTP traffic in port 80, excluding network overhead (Ethernet, IP, and TCP headers), but including HTTP headers for both queries, replies, and entity headers. The latter account for 7.9% of the HTTP contents.

The most important content types found in our trace are binaries, multimedia contents (JPEG, GIF, Windows Media Video, etc.), HTML, and plain text.

Figure 4.17 shows the savings in the amount of HTTP traffic shunted when using the BTL approach. In order to avoid sending commands to the device for very small entity bodies, we show the percentage of saved bytes as a function of the length of the smallest entity body required to justify a new BTL entry (X axis). This also covers the effects of packetization: The first bytes of an HTTP entity body are sent in the same packet that its entity header, and therefore cannot be processed at the shunt

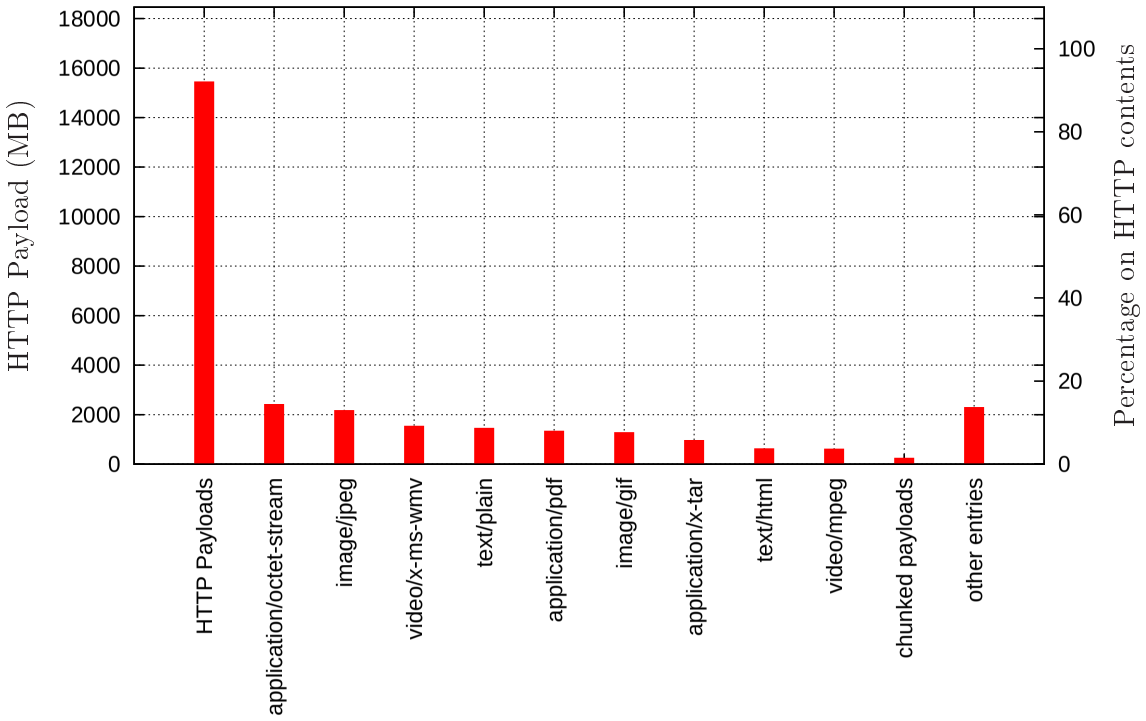


Figure 4.16: HTTP Breakup in trace tcp−1

device.

We have been careful to discount HTTP contents sent using chunked transfer coding from the savings [Fielding et al., 1999]. Chunked transfers are used to send dynamically produced content for which the total length is unknown until the end of the transfer. Therefore, there is no Content-Length field to use in the BTL field.

The operation described by the experiment is as follows: For a given HTTP connection from which the HTTP reply header is captured, if (a) the value of the Transfer-Encoding field is not chunked, and (b) the value of the Content-Length field is bigger than the threshold, then a forward entry is added to BTL-capable connection table in the shunt device. This entry is set to expire after Content-Length bytes are forwarded.

The upper three lines represent, respectively, the total amount of HTTP contents (including HTTP reply, response, and entity headers), the total amount of HTTP payloads (the transmitted data, discounting the HTTP reply and response headers), and the total amount of HTTP payloads (not including those sent using chunked coding).

The "Shunted HTTP Traffic" line shows the benefit of BTL assuming all non-chunked payloads bigger than the threshold are BTLed. If analyzing HTTP payloads is not of interest for the analysis, we could save around 90% of the total HTTP contents (and a proportional part of the corresponding Ethernet, IP, and TCP headers) by BTLing all payloads whose length is more than 5 KB.

The "*Skippable* Shunted HTTP Traffic" line shows the benefit of BTL assuming that only a subset of the HTTP non-chunked payloads are skipped using BTL (again, only those bigger than the threshold). The subset of skippable payloads includes those payloads where the MIME type in the value of the Content-Type field is audio, video, image, application. It excludes those payloads where the the MIME type is text or multipart/byteranges, which are instead shunted. (Payloads with MIME content type equal to multipart/byteranges are not skipped as they can reflect just about any type.) All other payloads are skipped.

This approach would save around 70% of the total HTTP contents (and a proportional part of the corresponding Ethernet, IP, and TCP headers) by BTLing all payloads in the skippable subset whose length is more than 5 KB.

Conclusions

The results of the BTL state that, from all HTTP contents, 75% of the bytes can be safely skipped by not shunting audio, image, video, and application payloads. If we consider the results from Section 4.6.3, where the traffic shunted (20% of the bytes) is 85% HTTP, a shunt device using BTL could provide a filtering ratio of 5% of the bytes.

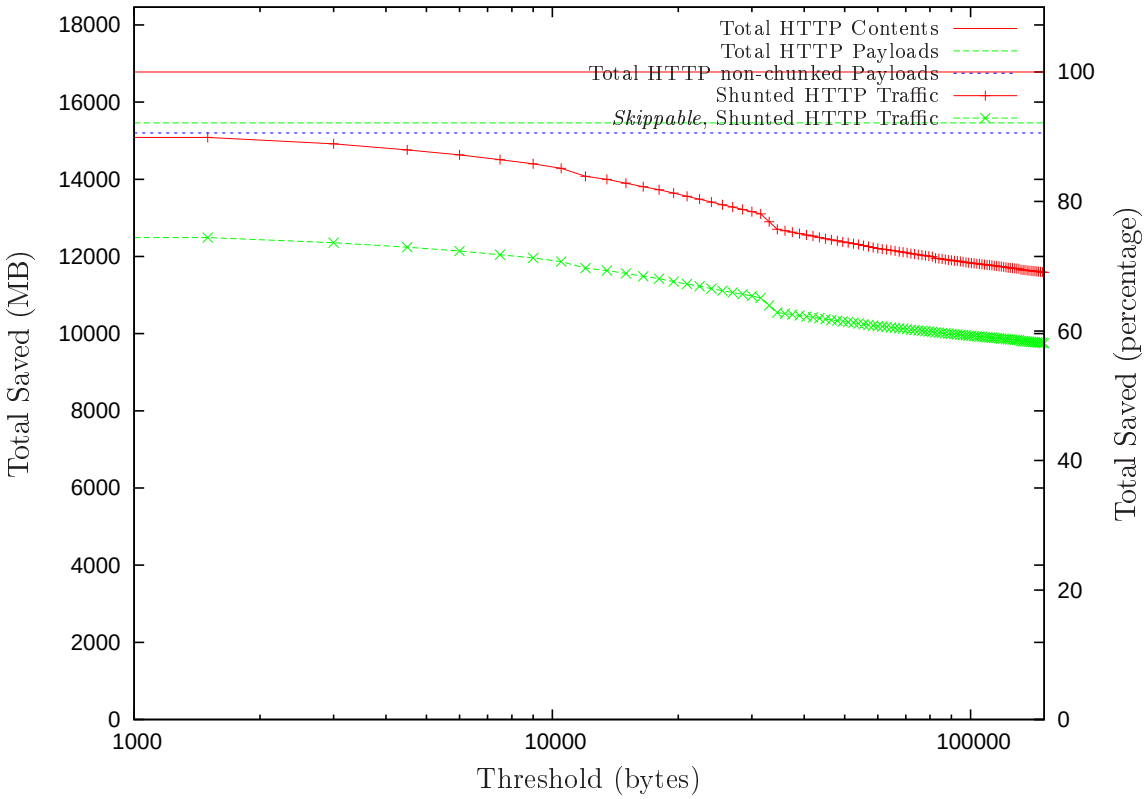


Figure 4.17: Benefit of BTL in HTTP Traffic

4.7.2 Shunting Other Analyzers

Another piece of future work is to keep modifying Bro analyzers to take advantage of the shunt capabilities. In the live experiment, for example, the first two candidate protocols to be modified this way are HTTPS and Microsoft NetBIOS, which account for 25% of the shunted bytes. In the *tcp-1* trace, the first two candidate protocols to be modified this way are HTTPS and SMTP, accounting for 9% of the shunted traffic before BTL, or 25% when reducing the HTTP traffic that BTL could process directly at the shunt device.

In the HTTPS case, the modification should be pretty simple, as the traffic is encrypted after the initial header, and the analyzer cannot process encrypted traffic without the proper key.

4.7.3 Evictions and Default Shunting

Default shunting may, in some rare cases, produce incorrect results. For example, consider a packet that is matched by entries in two tables. In one of the tables (T1), a high priority entry (E1) states that the packet must be forwarded. In the other table (T2), a lower priority entry (E2) states that the packet must be dropped. If both E1 and E2 exists, and assuming no other table or static filter have a higher-priority entry, the packet should be forwarded. Now, had E1 been evicted from T1, the packet would be dropped, instead of forwarded.

Note that, in order for this to be a problem, several facts must hold: (a) We expect

E1 to correspond to the connection table. The reason is that it is the table with a larger domain space, and therefore the most prone to evictions because of table space concerns. This means that E2 will be an entry in either the port or the address table. (b) E2 must request the packet to be forwarded or dropped. If E2 requests the packet to be shunted, then the shim will receive the packet, realize that an active entry (E1) was evicted, and reissue it. (c) E2 must have been issued after E1. Otherwise, the packet that caused E1 would have been forwarded or dropped because of E2, instead of reaching the analyzer, and E1 would have never been issued. (d) E1 and E2 must have been different yields.

An example of an error of this type would happen when E1 describes a connection that the analyzer wants shunted. Eventually, one of the hosts in the connection gets blacklisted/whitelisted (E2). Later, E1 is evicted for capacity reasons. At this moment, the analyzer loses visibility of the connection.

We believe that the right approach in the shunting architecture is to provide a mechanism to address it, and let the analyzer set the right policy. To address possible inconsistencies because of probabilistic device implementations, we provide the device with a mechanism to report to the analyzer when it decides to evict a table entry.

In order to prevent race conditions (the analyzer realizing that E1 was evicted and reissuing it, but only after E2 has dropped some packet), we plan to add a new eviction resolution message in the SHIP protocol. On receiving a request to add a new entry that would cause an eviction in one of its tables, the device will send back

to the analyzer an eviction resolution request. This request will include a list of the candidate entries for eviction. The analyzer will receive an event, and will answer stating which entry in the device should be evicted.

4.8 Conclusions

We have described a novel architecture to perform inline traffic processing in high-speed links using an off-the-shelf computer and a simple, special-purpose hardware device. The core of the architecture is the "shunt device," a network element that performs very simple packet processing (basically forwarding, dropping, or shunting packets to a software analyzer) at a very high speed.

Shunting provides fine-grained semantics to the analyzer using it, namely per-connection, per-address, and per-port processing. The goal of the architecture is that most packets are processed directly by the shunt device (either forwarded or dropped), while a small part are diverted through the intrusion-detection analyzer.

An added advantage of shunting is that, by default, all traffic is diverted to the analyzer. This allows the latter to perform intrusion prevention, blocking attack traffic before it can cause any harm.

We believe Shunting is an appealing architecture because it provides a large performance enhancement in return for a minimal additional mechanism. While the mechanisms are simple, they provide enough leverage for a NIPS to process high-speed links.

We have applied shunting to the intrusion detection arena, researched its performance benefits, and operated it in a real environment in order to understand its effects. We found that, when using shunting, the amount of traffic that the analyzer has to process gets reduced to one fifth in a real trace, and to 8% when considering BTL and a sensible configuration.

Shunting works specially well with some protocols, as SSH and FTP, reducing the cost of processing them to up to a factor of 20. It also opens the door to the analysis of similar protocols that pervade very high-speed links, as do Grids [Thain and Livny, 2003] in scientific laboratories.

In other protocols, as HTTP, the benefit of Shunting is compelling in the amount of filtered traffic, but the savings in resources are still poor. On the other hand, being able to perform full analysis in the HTTP traffic by just processing a 20% of the total traffic, plus the ability to take per-connection decisions in the hardware device, opens the door to fine-grained parallel approaches, where several NIDS share the responsibility of processing the shunt workload generated by a single device.

Bibliography

Aboba, B. (2001). Pros and cons of upper layer network access.

Agarwal, D., Gonzalez, J. M., Jin, G. and Tierney, B. (2003). An infrastructure for passive network monitoring of application data streams, *Proceedings of the Passive and Active Measurement Conference*.

Arramreddy, S. and Riley, D. (2002). PCI-X 2.0 white paper, *Technical report*, ServerWorks, Compaq.

Azar, Y., Broder, A., Karlin, A. and Upfal, E. (2000). Balanced allocations, *SIAM Journal on Computing* **29**(1): 180–200.

Bailey, M., Gopal, B., Pagels, M., Peterson, L. and Sarkar, P. (1994). Pathfinder: A pattern-based packet classifier, *Operating Systems Design and Implementation*, pp. 115–123.

Begel, A., McCanne, S. and Graham, S. L. (1999). BPF+: exploiting global data-flow optimization in a generalized packet filter architecture, *Proceedings of the Con-*

ference on Applications, Technologies, Architectures, and Protocols for Computer Communication, ACM Press, pp. 123–134.

Bellovin, S. M. (2002). A technique for counting natted hosts, *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*, ACM Press, pp. 267–272.

Bernaschi, M., Gabrielli, E. and Mancini, L. V. (2000). Operating system enhancements to prevent the misuse of system calls, *CCS '00: Proceedings of the 7th ACM Conference on Computer and Communications Security*, ACM Press, New York, NY, USA, pp. 174–183.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM* **13**(7): 422–426.

Bos, H., de Bruijn, W., Cristea, M., Nguyen, T. and Portokalidis, G. (2004). FFPF: Fairly fast packet filters, *Proceedings of OSDI'04*.

Braden, R. T. (1989). RFC 1122: Requirements for Internet hosts — communication layers.

Brewer, J. and Sekel, J. (2004). PCI express technology, *Technical report*, Dell.

Carter, J. and Wegman, M. (1979). Universal classes of hash functions, *Journal of Computer and Systems Sciences*, Vol. 18.

Claffy, K. C., Polyzos, G. C. and Braun, H.-W. (1993). Application of sampling methodologies to network traffic characterization, *Conference Proceedings on Communications Architectures, Protocols and Applications*, ACM Press, pp. 194–203.

Clark, D., Jacobson, V., Romkey, J. and Salwen, H. (1989). An analysis of tcp processing overheads, *IEEE Communication Magazine* **27**(2): 23–29.

Cleary, J., Donnelly, S., Graham, I., McGregor, A. and Pearson, M. (2000). Design principles for accurate passive measurement, *Proceedings of the Passive and Active Measurement Conference*.

Compaq (1999). PCI-X: An evolution of the pci bus, *Technical report*, Compaq.

Coppens, J., den Berghe, S. V., Bos, H., Markatos, E., Turck, F. D., Oslebo, A. and Ubik, S. (2003). Scampi - a scaleable and programmable architecture for monitoring gigabit networks, *Proceedings of E2EMON Workshop*.

Coppens, J., Markatos, E., Novotny, J., Polychronakis, M., Smotlacha, V. and Ubik, S. (2004). Scampi - a scaleable monitoring platform for the internet, *Proceedings of the 2nd International Workshop on Inter-Domain Performance and Simulation (IPS 2004)*.

Cristea, M. and Bos, H. (2004). A compiler for packet filters, *Proceedings of ASCI'04*.

Cristea, M., de Bruijn, W. and Bos, H. (2005). Fpl-3: towards language support for distributed packet processing, *Proceedings of IFIP Networking*.

Crosby, S. and Wallach, D. (2003). Denial of service via algorithmic complexity attacks, *Proceedings of the 12th USENIX Security Symposium*, pp. 29–44.

Crovella, M. (2001). Performance evaluation with heavy tailed distributions, *JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, London, UK, pp. 1–10.

Crovella, M. and Bestavros, A. (1996). Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes, *Proceedings of SIGMETRICS'96: The ACM International Conference on Measurement and Modeling of Computer Systems.*, Philadelphia, Pennsylvania. Also, in Performance evaluation review, May 1996, 24(1):160-169.

Currid, A. (2004). TCP offload to the rescue, *Queue* **2**(3): 58–65.

Degioanni, L. and Varenni, G. (2004). Introducing scalability in network measurement: Toward 10 Gbps with commodity hardware, *IMC '04: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, ACM Press, New York, NY, USA, pp. 233–238.

Denning, D. E. (1987). An intrusion-detection model, *IEEE/ACM Transactions on Software Engineering* **13**(2): 222–232.

Deri, L. (2003). Passively monitoring networks at gigabit speeds using commodity

hardware and open source software, *Proceedings of the Passive and Active Measurement Conference*.

Dreger, H. (2004). Personal communication.

Dreger, H., Feldmann, A., Paxson, V. and Sommer, R. (2004). Operational experiences with high-volume network intrusion detection, *Proceedings of CCS*.

Duffield, N., Lund, C. and Thorup, M. (2002). Properties and prediction of flow statistics from sampled packet streams, *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*, ACM Press, pp. 159–171.

Duffield, N., Lund, C. and Thorup, M. (2003). Estimating flow distributions from sampled flow statistics, *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ACM Press, pp. 325–336.

Dykstra, P. (1999). Gigabit ethernet jumbo frames. and why you should care. White Paper.

Engler, D. R. and Kaashoek, M. F. (1996). DPF: Fast, flexible message demultiplexing using dynamic code generation, *SIGCOMM*, pp. 53–59.

Estan, C. and Varghese, G. (2002). New directions in traffic measurement and accounting, *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ACM Press, pp. 323–336.

Estan, C., Savage, S. and Varghese, G. (2003). Automatically inferring patterns of resource consumption in network traffic, *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ACM Press, pp. 137–148.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. (1999). RFC 2616: Hypertext transfer protocol – HTTP/1.1. Status: INFORMATIONAL.

Forrest, S., Hofmeyr, S. A., Somayaji, A. and Longstaff, T. A. (1996). A sense of self for Unix processes, *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society Press, pp. 120–128.

Garfinkel, T. and Rosenblum, M. (2003). A virtual machine introspection based architecture for intrusion detection, *Proceedings of the Network and Distributed Systems Security Symposium*.

Gil, T. M. and Poletto, M. (2001). MULTOPS: A Data-Structure for bandwidth attack detection, *Proceedings of the 10th USENIX Security Symposium*, pp. 23–38.

Gilder, G. (2000). *TELECOSM: How Infinite Bandwidth will Revolutionize Our World*, Free Press.

Giovanni, C. (2001). Fun with packets: Designing a stick, *Technical report*, Endeavor Systems.

Handley, M., Kreibich, C. and Paxson, V. (2001). Network intrusion detection: Evasion, traffic normalization, end end-to-end protocol semantics, *Proceedings of the 9th USENIX Security Symposium*.

Iannaccone, G., Diot, C., Graham, I. and McKeown, N. (2001). Monitoring very high speed links, *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pp. 267–271.

Intel (2005). Intel(r) network infrastructure processors: Extending intelligence in the network.

Ioannidis, S., Anagnostakis, K., Ioannidis, J. and Keromytis, A. (2002). xpf: packet filtering for lowcost network monitoring, *Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR)*, pp. 121–126.

Jacobson, V. and Karels, M. J. (1988). Congestion avoidance and control, *ACM Computer Communication Review; Proceedings of the SIGCOMM '88 Symposium in Stanford, CA, August, 1988* **18, 4**: 314–329.

Jacobson, V., Braden, R. and Borman, D. (1992). RFC 1323: TCP extensions for high performance.

Jelena, P. and Greg, M. (2002). Attacking DDoS at the source, *ICNP '02: Proceedings of the 10th IEEE International Conference on Network Protocols*, IEEE Computer Society, Washington, DC, USA, pp. 312–321.

Karagiannis, T., Broido, A., Faloutsos, M. and Claffy, K. (2004). Transport layer identification of p2p traffic, *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pp. 121–134.

Karp, R. M., Luby, M. and auf der Heide, F. M. (1992). Efficient PRAM simulation on a distributed memory machine, pp. 318–326.

Kay, J. and Pasquale, J. (1993). The importance of non-data touching processing overheads in tcp/ip, *SIGCOMM '93: Conference Proceedings on Communications Architectures, Protocols and Applications*, ACM Press, New York, NY, USA, pp. 259–268.

Kleinpaste, K., Steenkiste, P. and Zill, B. (1995). Software support for outboard buffering and checksumming, *SIGCOMM '95: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ACM Press, New York, NY, USA, pp. 87–98.

Ko, C., Ruschitzka, M. and Levitt, K. (1997). Execution monitoring of security-critical programs in distributed systems: a specification-based approach, *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, IEEE Computer Society, Washington, DC, USA, p. 175.

Kreibich, C., Warfield, A., Crowcroft, J., Hand, S. and Pratt, I. (2005). Using packet symmetry to curtail malicious traffic, *Proceedings of the Fourth Workshop on Hot Topics in Networks (HotNets-IV) (to appear)*, ACM SIGCOMM.

Kruegel, C., Valeur, F., Vigna, G. and Kemmerer, R. (2002). Stateful intrusion detection for high-speed networks, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 285–294.

Kumar, A., Paxson, V. and Weaver, N. (2005). Exploiting underlying structure for detailed reconstruction of an internet-scale event, *Proceedings of the ACM Internet Measurement Conference*.

Lee, J., Gunter, D., Tierney, B., Allcock, B., Bester, J., Bresnahan, J. and Tuecke, S. (2001). Applied techniques for high bandwidth data transfers across wide area networks, *Proceedings of International Conference on Computing in High Energy and Nuclear Physics*.

Lee, W., Cabrera, J., Thomas, A., Balwalli, N., Saluja, S. and Zhang, Y. (2002). Performance adaptation in real-time intrusion detection systems, *RAID*, pp. 252–273.

Linhart, C., Klein, A., Heled, R. and Orrin, S. (2005). HTTP request smuggling, *Technical report*, Watchfire.

Lohr, S. (1999). *Sampling: Design & Analysis*, Thomson Learning.

Markatos, E. (2005). Scampi detailed architecture design.

McCanne, S. and Jacobson, V. (1993). The BSD packet filter: A new architecture for user-level packet capture, *USENIX Winter*, pp. 259–270.

Medina, A., Fraleigh, C., Taft, N., Bhattacharyya, S. and Diot, C. (2002). A Taxonomy of IP Traffic Matrices, *SPIE ITCOM: Scalability and Traffic Control in IP Networks II*, Boston.

Mills, C., Hirsh, D. and Ruth, G. R. (1991). RFC 1272: Internet accounting: Background.

Mitzenmacher, M. and Upfal, E. (2005). *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge University Press.

Mogul, J. (1990). Efficient use of workstations for passive monitoring of local area networks, *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, ACM Press, pp. 253–263.

Mogul, J. C. (2003). Tcp offload is a dumb idea whose time has come., *HotOS*, pp. 25–30.

Mogul, J. C. and Deering, S. E. (1990). RFC 1191: Path MTU discovery.

Mogul, J. C. and Ramakrishnan, K. K. (1997). Eliminating receive livelock in an interrupt-driven kernel, *ACM Transactions on Computer Systems* **15**(3): 217–252.

Mogul, J., Rashid, R. and Accetta, M. (1987). The packet filter: An efficient mechanism for user-level network code, *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, Vol. 21, pp. 39–51.

Moore, A., Hall, J., Kreibich, C., Harris, E. and Pratt, I. (2003a). Architecture of a network monitor, *Proceedings of the Passive and Active Measurement Conference*.

Moore, D. and Shannon, C. (2004). The spread of the witty worm, *IEEE Security and Privacy* **2**(4): 46–50.

Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S. and Weaver, N. (2003b). Inside the slammer worm, *IEEE Security and Privacy* **1**(4): 33–39.

Morris, R., Kohler, E., Jannotti, J. and Kaashoek, M. F. (1999). The click modular router, *Symposium on Operating Systems Principles*, pp. 217–231.

Morrison, J. (1985). EA IFF 85: Standard for interchange format files, *Technical report*, Electronic Arts.

Mukherjee, B., Heberlein, L. and Levitt, K. (1994). Network intrusion detection, *IEEE Network* pp. 26–41.

Nguyen, T., Cristea, M., de Bruijn, W. and Bos, H. (2004). Scalable network monitors for high-speed links: A bottom-up approach, *Proceedings of IPOM'04*.

Park, S. K. and Miller, K. W. (1988). Random number generators: good ones are hard to find, *Communications of the ACM* **31**(10): 1192–1201.

Patterson, D. A. and Hennessy, J. (2004). *Computer Organization and Design*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R. and Yelick, K. (1997). A case for intelligent RAM, *IEEE Micro* **17**(2): 34–44.

Patton, S., Yurcik, W. and Doss, D. (2001). An achilles' heel in signature-based IDS: Squealing false positives in SNORT, *RAID*.

Paxson, V. (1994). Empirically derived analytic models of wide-area TCP connections, *IEEE/ACM Transactions on Networking* **2**(4): 316–336.

Paxson, V. (1999). Bro: A system for detecting network intruders in real-time, *Computer Networks (Amsterdam, Netherlands: 1999)* **31**(23–24): 2435–2463.

Paxson, V. and Floyd, S. (1995). Wide area traffic: The failure of poisson modeling, *IEEE/ACM Transactions on Networking* **3**(3): 226–244.

Pennington, A., Strunk, J., Griffin, J., Soules, C., Goodson, G. and Ganger, G. (2003). Storage-based intrusion detection: Watching storage activity for suspicious behavior, *Proceedings of the USENIX Security Symposium*.

Postel, J. (1981a). RFC 791: Internet Protocol.

Postel, J. (1981b). RFC 793: Transmission control protocol.

Postel, J. and Reynolds, J. (1985). RFC 959: File transfer protocol.

Ptacek, T. H. and Newsham, T. N. (1998). Insertion, evasion, and denial of

service: Eluding network intrusion detection, *Technical report*, Secure Networks, Inc., Calgary, Alberta, Canada.

Rivest, R. (1992). RFC 1321: The MD5 message-digest algorithm. Status: INFORMATIONAL.

Roesch, M. (1999). Snort: Lightweight intrusion detection for networks, *Proceedings of the 13th USENIX Conference on System Administration*, USENIX Association, pp. 229–238.

Schneider, F. (2004). Analyse der leistung von BPF und libpcap in Gigabit-Ethernet Umgebungen, *Technical report*, Technische Universitat Munchen.

Schneier, B. (1995). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, Inc., New York, NY, USA.

Shankar, U. and Paxson, V. (2003). Active mapping: Resisting NIDS evasion without altering traffic, *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, IEEE Computer Society, p. 44.

Shannon, C., Moore, D. and Claffy, K. C. (2002). Beyond folklore: Observations on fragmented traffic, *IEEE/ACM Transactions on Networking* **10**(6): 709–720.

Sommer, R. and Paxson, V. (2005). Exploiting independent state for network intrusion detection, *The IEEE Annual Computer Security Applications Conference (ACSAC'05)*.

Song, D. (2001). Fragroute source code, *Technical report*, www.monkey.org.

Staniford, S., Hoagland, J. A. and McAlerney, J. M. (2002a). Practical automated detection of stealthy portscans, *Journal of Computer Security* **10**(1-2): 105–136.

Staniford, S., Paxson, V. and Weaver, N. (2002b). How to 0wn the internet in your spare time, *Proceedings of the 11th USENIX Security Symposium*.

Thain, D. and Livny, M. (2003). *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann.

van der Merwe, J., Caceres, R., Chu, Y. and Sreenan, C. (2000). mmdump: a tool for monitoring internet multimedia traffic, *SIGCOMM Computer Communications Review*, Vol. 30, pp. 48–59.

Wagner, D. and Dean, D. (2001). Intrusion detection via static analysis, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 156–169.

Weaver, N., Ellis, D., Staniford, S. and Paxson, V. (2004). Worms vs. perimeters: The case for hard-LANs, *12th Annual Proceedings of IEEE Hot Interconnects (HotI-12)*, Stanford, CA.

Wood, P. (2004). *libpcap-mmap*: Available at http://public.lanl.gov/cpw/.

Xu, K., Zhang, Z.-L. and Bhattacharyya, S. (2005). Profiling internet backbone traffic: Behavior models and applications, *SIGCOMM '05: Proceedings of the 2005 Con-*

ference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM Press, New York, NY, USA, pp. 169–180.

Ylonen, T. (1996). SSH – Secure Login Connections over the Internet, *Proceedings of the 6th USENIX Security Symposium*, San Jose, CA, USA, pp. 37–42.

Yuhara, M., Bershad, B. N., Maeda, C. and Moss, J. E. B. (1994). Efficient packet demultiplexing for multiple endpoints and large messages, *USENIX Winter*, pp. 153–165.

Zhang, Y. and Paxson, V. (2000a). Detecting backdoors, *Proceedings of the 9th USENIX Security Symposium*, pp. 157–170.

Zhang, Y. and Paxson, V. (2000b). Detecting stepping stones, *Proceedings of the 9th USENIX Security Symposium*, pp. 171–184.

Appendix A

Secondary Path Details

A.1 Generic Algorithm for Detecting Interactive Backdoors

This Section describes the implementation of the Generic Algorithm for Detecting Interactive Backdoors described in [Zhang and Paxson, 2000a] using the Secondary Path.

Figure A.1 shows the generic analyzer code.

Note that the filter we propose is slightly different from the one suggested by [Zhang and Paxson, 2000a]. We got rid of the packets with zero-length payload, which correspond to machine-driven transport-layer acknowlegments, but are common enough as to affect significantly the performance of the Secondary Path-based solution.

```
global generic_sig_filter =
  "tcp and
    ((ip[2:2] - ((ip[0]&0x0f)<<2) - (tcp[12]>>2)) <= 20) and
    ((ip[2:2] - ((ip[0]&0x0f)<<2) - (tcp[12]>>2)) > 0)";


const interconn_min_num_pkts = 10 &redef; # min num of pkts sent
const interconn_min_alpha = 0.2 &redef; # minimum required alpha
const interconn_min_gamma = 0.2 &redef; # minimum required gamma


function comp_gamma(s: conn_info): double
  {
  return s$N >= interconn_min_num_pkts ?
    (1.0 * (s$S - s$G - 1)) / s$N : 0.0;
  }


function comp_alpha(s: conn_info) : double
  {
  return ( s$short_intervals > 0 ) ?
    (1.0 * s$large_intervals / s$short_intervals) : 0.0;
  }


function is_interactive_endp(s: conn_info): bool
  {
  # Criteria 1: num_pkts >= interconn_min_num_pkts.
  if ( s$N < interconn_min_num_pkts )
    return F;

  # Criteria 2: gamma >= interconn_min_gamma.
  if ( comp_gamma(s) < interconn_min_gamma )
    return F;

  # Criteria 3: alpha >= interconn_min_alpha.
  if ( comp_alpha(s) < interconn_min_alpha )
    return F;

  return T;
  }
```

Figure A.1: Generic Backdoor Detector Implementation

```
function estimate_gap(gap: count): count
  {
 return (gap + interconn_default_pkt_size - 1) / interconn_default_pkt_size;
  }


function interval_is_short(t: interval): bool
  {
 return (interconn_min_interarrival <= t) && (t <= interconn_max_interarrival);
  }


event backdoor_generic_sig(filter: string, pkt: pkt_hdr)
  {
  # get rid of traffic in well-known ports
  if ( interconn_ignore_standard_ports &&
       (pkt$tcp$sport in interconn_standard_ports ||
       pkt$tcp$dport in interconn_standard_ports) )
    {
    return;
    }

  # create the connection id
  local id = [
    $orig_h = pkt$ip$src, $orig_p = pkt$tcp$sport,
    $resp_h = pkt$ip$dst, $resp_p = pkt$tcp$dport];
  local payload_length = pkt$ip$len - pkt$ip$hl - pkt$tcp$hl;
  local seq = pkt$tcp$seq + payload_length;

  # if inexistent connection => create blank entry
  if ( id !in interconn_conns )
    {
    interconn_conns[id] = [
      $S = 1,
      $N = 1,
      $G = 0,
      $top_seq = seq,
      $last_ts = network_time(),
      $short_intervals = 0,
      $large_intervals = 0,
      $interactive = INTERCONN_UNKNOWN];
    }
```

Figure A.2: Generic Backdoor Detector Implementation (cont.)

```
  else
    {
    # we got a (small) packet
    ++interconn_conns[id]$S;
    ++interconn_conns[id]$N;

    local top_seq = interconn_conns[id]$top_seq + payload_length;

    if ( top_seq != 0 && top_seq < seq )
      {
      # there's been a gap in this connection
      ++interconn_conns[id]$G;
      #interconn_conns[id]$N += estimate_gap(seq - top_seq);
    interconn_conns[id]$N = interconn_conns[id]$N + estimate_gap(seq - top_seq);
      }

    interconn_conns[id]$top_seq = seq;

    if ( interval_is_short(network_time() - interconn_conns[id]$last_ts) )
      ++interconn_conns[id]$short_intervals;
    else
      ++interconn_conns[id]$large_intervals;
    interconn_conns[id]$last_ts = network_time();
    }

  if ( ! is_interactive_endp(interconn_conns[id])  )
    return;

  log_interconn(id);
  }

redef secondary_filters += {
  [generic_sig_filter] = backdoor_generic_sig,
};
```

Figure A.3: Generic Backdoor Detector Implementation (cont.)

Appendix B

Shunt Details

B.1 Shunt Interconnect Protocol (SHIP) Details

This Section describes in detail each of the SHIP messages.

For the control message type, SHIP uses 4-byte identifiers with a human readable sequence. For example, the type field for the OPEN message is 0x4f50454e, whose ASCII translation is the string "OPEN".

For the control message payloads, SHIP uses two different structures: Messages with fixed contents use a fixed structure, while messages with variable contents divide their payload in chunks, each composed of a 4-byte identifier, a 4-byte data size (always a multiple of 4 for alignment reasons), and the data itself. The 4 byte identifier is again a human readable sequence of four characters, such as "FILE" or "VERS".

Figure B.1 shows an example of a variable-content SHIP payload. The payload

is composed of two (variable) fields, one containing a 4-byte integer with a filter priority, and the other containing a 10-byte long string (plus 2 bytes of padding) with the expression to be used for the static forward filter.

Note that this structure is very similar to the one used in some generic file formats, as the Interchange File Format Morrison [1985].

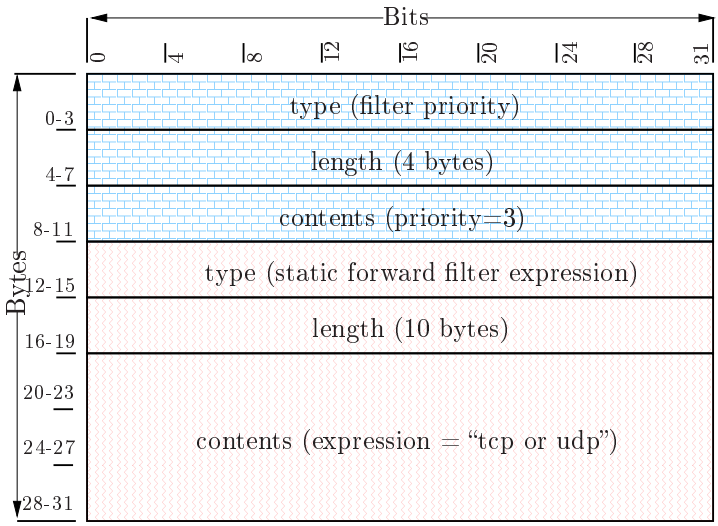


Figure B.1: Shunt Interconnect Protocol Variable Payload Example

B.1.1 ACK Messages

ACK messages just acknowledge a packet sent from the other side. Their only parameter is the message identifier of the acknowledged packet. Its payload is fixed, just 4 bytes with the such message identifier.

B.1.2 Device-Ready Messages

Device-Ready messages are sent from the device to the shim to report it is alive and waiting for initialization. The only parameter is a unique device identifier. Its payload is fixed, 4 bytes with the device identifier.

B.1.3 Open Messages

Open messages are sent from the shim to the device to put the device in working mode. Its payload is variable, including some or all of the items in Table B.1.

field	type	explanation
filter string(forward)	string	tcpdump expression to be used as static forward filter
filter string(drop)	string	tcpdump expression to be used as static drop filter
filter string(shunt)	string	tcpdump expression to be used as static shunt filter
filter priority(forward)	integer	priority of the static forward filter
filter priority(drop)	integer	priority of the static drop filter
filter priority(shunt)	integer	priority of the static shunt filter
filter sample	3 bits	sampling ratio of the three static filters
default sample	3 bits	default sampling ratio
failsafe mode	Boolean	fail-safe mode (fail-open or fail-close)

Table B.1: SHIP Open Variable Contents

B.1.4 Capabilities Messages

Device Capabilities messages are sent from the device to the shim on reception of a Open Message. Its payload is variable, including some or all of the items in Table B.2.

B.1.5 Close Messages

Close messages are sent from the shim to the device to put it back into fail-safe mode, or from the device to the shim to report that an internal problem is too important to keep in working mode, and that therefore the device is moving into fail-safe state. The payload is fixed, and there are no arguments.

B.1.6 Reset Messages

Reset messages are sent from the shim to the device to instruct it to reset all its tables and/or accounting information. The payload is fixed, and it consists of three 4-byte values, each representing a Boolean. The first Boolean states whether the device must carry out a hard reset, which removes all the table entries, resets all the device statistics, and cleans up the retransmission buffer. The second Boolean states whether the device must reset all its statistics. The third Boolean states whether the device must remove all the table entries.

B.1.7 Error Messages

Error messages are sent from the device to the shim to report of an error. The payload is fixed, and it consists of a 4-byte integer with a human readable sequence

field	type	explanation
version	integer	device version identifier
nics	integer list	unique identifiers for each of the device's network taps

Table B.2: SHIP Device Capabilities Variable Contents

of four characters that expresses the error that happened in the device. There are no valid values defined so far.

B.1.8 Status-Request Messages

Status-Request messages are sent from the shim to the device to request the contents of the device's tables. The payload is fixed, and it consists of a 4-byte value representing a Boolean that states whether the status response must occur just once or be fired periodically.

Status-Request and Status-Response messages are used in the synchronization method described in Section 4.5.2.

B.1.9 Status-Response Messages

Status-Response messages include a dump of the contents of the sender's (device or shim) tables. They may be sent (a) from the device to the shim as a response to a Status-Request message, or (b) from the shim to the device in order to populate the latter's tables. The payload is variable, and it consists of a set of chunks, each composed of a 4-byte table identifier, a 4-byte data size (again a multiple of 4 for alignment reasons), and the entry's index and yield. There are 4 different table identifiers, as described in Table B.3.

Figure B.2 shows an example of the encoding of an entry in a SHIP Status-Response message. Note that the bits are not encoding for efficiency (for example, the forth

and back forward and shunt fields use 32 bits each, while there is only one bit being used).

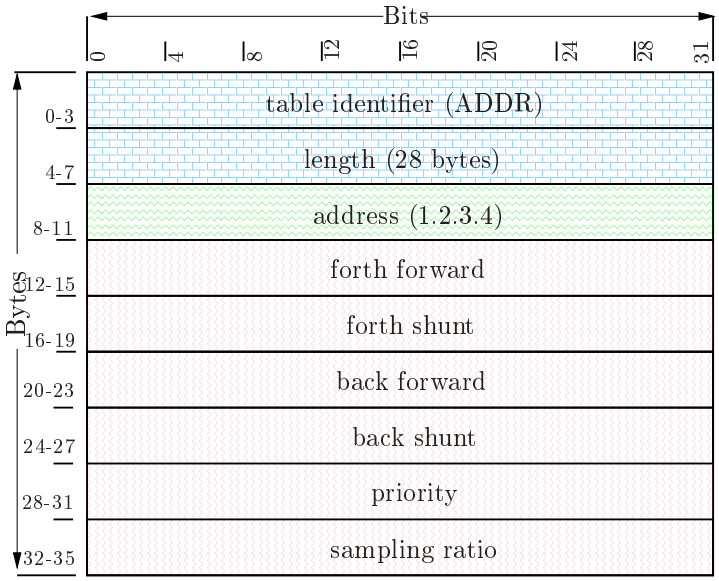


Figure B.2: SHIP Status Response Entry Example

B.1.10 Statistics-Request Messages

Statistics-Request messages are sent from the shim to the device to request the device's operation statistics. These statistics include the amount of packet and bytes, forwarded, shunted, or dropped, for different reasons: connection ID matching the connection table, source or destination address matching the address table, source or

identifier	explanation
CONN	an entry in the connection table
ADDR	an entry in the address table
PORT	an entry in the port table
DONE	last packet in the synchronization process

Table B.3: SHIP Status Response Table Types

destination port matching the port table, packet matching any of the 3 static filters, or default shunting.

The request payload is fixed, and it consists of a 4-byte value representing a Boolean that states whether the statistics response must occur just once or be fired periodically.

B.1.11 Statistics-Response Messages

Statistics-Response messages include a dump of the contents of the device's operation statistics.

The payload is fixed, and it consists of a the values mentioned in Section B.1.10.

B.1.12 Associate Connection Messages

Associate Connection messages are sent from the shim to the device to add a new entry in the device's connection table.

The payload is fixed, and it consists of a 20-byte value representing a 104-bit connection identifier (source and destination address and port, plus transport port), followed by a 28-byte value representing the 10-bit yield. The latter is composed of 1 bit for the forward decision in the forth direction, 1 bit for the forward decision in the back direction, 1 bit for the shunt decision in the forth direction, 1 bit for the shunt decision in the back direction, 3 bits for the entry priority, and 3 bits for the sampling ratio.

B.1.13 Deassociate Connection Messages

Deassociate Connection messages are sent from the shim to the device to remove an entry from the device's connection table, or from the device to the shim to report an entry being evicted from the connection table.

The payload is fixed, and it consists of a 20-byte value representing a 104-bit connection identifier (source and destination address and port, plus transport port), followed by a 4-byte value representing the reason why the entry was evicted (this value is unused when the message is sent from the shim to the device).

B.1.14 Associate Address Messages

Associate Address messages are sent from the shim to the device to add a new entry in the device's address table.

The payload is fixed, and it consists of a 4-byte value representing an IP address, followed by a 28-byte value representing the 10-bit yield. The latter is composed of 1 bit for the forward decision in the forth direction, 1 bit for the forward decision in the back direction, 1 bit for the shunt decision in the forth direction, 1 bit for the shunt decision in the back direction, 3 bits for the entry priority, and 3 bits for the sampling ratio.

B.1.15 Deassociate Address Messages

Deassociate Address messages are sent from the shim to the device to remove an entry from the device's address table, or from the device to the shim to report an entry being evicted from the address table.

The payload is fixed, and it consists of a 4-byte value representing an IP address, followed by a 4-byte value representing the reason why the entry was evicted (this value is unused when the message is sent from the shim to the device).

B.1.16 Associate Port Messages

Associate Port messages are sent from the shim to the device to add a new entry in the device's port table.

The payload is fixed, and it consists of an 8-byte value representing a transport-layer protocol and port, followed by a 28-byte value representing the 10-bit yield. The latter is composed of 1 bit for the forward decision in the forth direction, 1 bit for the forward decision in the back direction, 1 bit for the shunt decision in the forth direction, 1 bit for the shunt decision in the back direction, 3 bits for the entry priority, and 3 bits for the sampling ratio.

B.1.17 Deassociate Port Messages

Deassociate Port messages are sent from the shim to the device to remove an entry from the device's port table, or from the device to the shim to report an entry being

evicted from the port table.

The payload is fixed, and it consists of an 8-byte value representing a transport-layer protocol and port, followed by a 4-byte value representing the reason why the entry was evicted (this value is unused when the message is sent from the shim to the device).

B.2 Shim Application Programming Interface

This Section describes in detail the Shunting API, as exported to Bro.

B.2.1 shunt_open() Function

The shunt_open() function can be used by the analyzer to request the shim to open the device. It has no parameters.

B.2.2 shunt_close() Function

The shunt_close() function can be used by the analyzer to request the shim to close the device. It has no parameters.

B.2.3 shunt_reset() Function

The shunt_reset() function can be used by the analyzer to request the shim to reset the device. It has three Boolean parameters: The first Boolean states whether

the device must carry out a hard reset, which removes all the table entries, resets all the device statistics, and cleans up the retransmission buffer. The second Boolean states whether the device must reset all its statistics. The third Boolean states whether the device must remove all the table entries.

B.2.4 shunt_drop_packet() Function

The shunt_drop_packet() function can be used by the analyzer to request that the packet currently being analyzer be dropped after injection, without further filtering.

B.2.5 shunt_inject_packet() Function

The shunt_inject_packet() function can be used by the analyzer to request that the packet currently being analyzer be injected back into the wire, without further filtering.

B.2.6 shunt_get_status() Function

The shunt_get_status() function can be used by the analyzer to request to start the table synchronization mechanism, so that the device table contents are sent to the shim.

B.2.7 shunt_status_event() Event

The shunt_status_event() event is fired in the analyzer for every entry coming from the device that is received by the shim during synchronization.

B.2.8 shunt_get_statistics() Function

The shunt_get_statistics() function can be used by the analyzer to request that the shim obtains the device's operation statistics.

B.2.9 shunt_statistics_event() Event

The shunt_statistics_event() event is fired in the analyzer once it has received the device operation statistics.

B.2.10 shunt_associate_conn() Function

The shunt_associate_conn() function can be used by the analyzer to add an entry in the connection table (both at the shim and at the device). The function parameters are the connection identifier (source and destination IP addresses and transport-layer ports, plus the transport protocol), the forth action (*forward*, *drop*, *shunt*, or "forward and shunt"), the back action (same possibilities), the priority of the entry, and the sampling ratio.

B.2.11 shunt_deassociate_conn() Function

The shunt_deassociate_conn() function can be used by the analyzer to remove an entry from the connection table (both at the shim and at the device). The function parameter consists of the connection identifier (source and destination IP addresses and transport-layer ports).

B.2.12 shunt_associate_addr() Function

The shunt_associate_addr() function can be used by the analyzer to add an entry in the address table (both at the shim and at the device). The function parameters are the IP address, the forth action (*forward*, *drop*, *shunt*, or "forward and shunt"), the back action (same possibilities), the priority of the entry, and the sampling ratio.

B.2.13 shunt_deassociate_addr() Function

The shunt_deassociate_addr() function can be used by the analyzer to remove an entry from the address table (both at the shim and at the device). The function parameter consists of the IP address.

B.2.14 shunt_associate_port() Function

The shunt_associate_port() function can be used by the analyzer to add an entry in the port table (both at the shim and at the device). The function parameters are the transport-layer port, the transport protocol, the forth action (*forward*, *drop*,

shunt, or "forward and shunt"), the back action (same possibilities), the priority of the entry, and the sampling ratio.

B.2.15 shunt_deassociate_port() Function

The shunt_deassociate_port() function can be used by the analyzer to remove an entry from the port table (both at the shim and at the device). The function parameter consists of the transport-layer port and the transport protocol.

B.2.16 shunt_evict_conn_event() Event

The shunt_evict_conn_event() event is fired by the shim every time it receives a report from the shim stating that it had to evict an entry in the connection table.

The event parameter consists of the connection identifier (source and destination IP addresses and transport-layer ports, plus the transport protocol).

B.2.17 shunt_evict_addr_event() Event

The shunt_evict_addr_event() event is fired by the shim every time it receives a report from the shim stating that it had to evict an entry in the address table.

The event parameter consists of the entry's IP address.

B.2.18 shunt_evict_port_event() Event

The shunt_evict_port_event() event is fired by the shim every time it receives a report from the shim stating that it had to evict an entry in the port table.

The event parameter consists of the entry's transport protocol and the transport-layer port.

B.2.19 shunt_inconsistent_conn_event() Event

The shunt_inconsistent_conn_event() event is fired by the shim every time it receives a shunted packet that was processed incorrectly at the device because of a connection table entry. A typical case is when, for space reasons, the device must have removed an entry in its connection table, and a packet that should have been forwarded is instead shunted. As soon as the shim receives the packet, it processes the packet through its tables and static filters. If it finds that the packet should have been dealt with differently in the device because of a connection table entry, this event is issued.

The analyzer response may be, for example, to reissue the entry again. Note that reissuing the connection entry must cause yet another table eviction, which eventually may cause another shunt_inconsistent_conn_event() event. If the number of active connections fighting for the same device table frames is too large, there would be thrashing. It is the analyzer's responsibility to detect thrashing and react adequately.

The only event parameter is the connection identifier (source and destination IP

addresses and transport-layer ports, plus the transport protocol).

B.2.20 shunt_inconsistent_addr_event() Event

The shunt_inconsistent_addr_event() event is fired by the shim every time it receives a shunted packet that was processed incorrectly at the device because of an address table entry.

The only event parameter is the IP address.

B.2.21 shunt_inconsistent_port_event() Event

The shunt_inconsistent_port_event() event is fired by the shim every time it receives a shunted packet that was processed incorrectly at the device because of an port table entry.

The event parameters consist of the entry's transport protocol and the transport-layer port.

B.3 Ethertype Field Information Packing

This Section describes the remapping of the Ethernet Type (ethertype) field used in order to pack per-shunted packet information in data messages sent between shim and device, and described in Section 4.5.3.

Figure B.3 shows the remapping of the 16 bit Ethernet Header's Type Field.

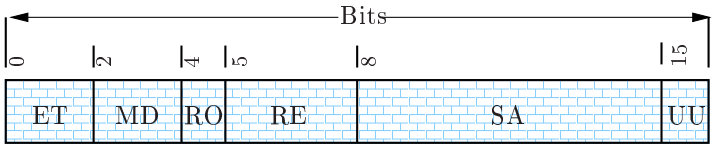


Figure B.3: 16 bit Ethernet Header Type Field Remapping

Table B.4 describes the remapping in depth.

code	size (bits)	explanation
ET	2	*Real Ethernet Type Encoding* Possible values are 00 (invalid)[1], 01 (IP), 10 (ARP), and 11 (Reverse ARP)
MD	2	*Main Decision* Possible values are 00 (invalid), 01 (packet was forwarded), 10 (packet was dropped), 11 (packet was shunted)[2]
RO	1	*Routing Information* Possible values are 0 and 1, depending on whether the packet was originally received in the device's first or second network interface, respectively
RE	3	*Reason Why the Packet Was Marked as MD* Possible values are 000 (invalid), 001 (match in connection table), 010 (match in address table for the source address), 011 (match in address table for the destination address), 100 (match in port table for the source port), 101 (match in port table for the destination port), 110 (match in any of the three static filters), or 111 (default decision)
SA	7	*Sampling Information* Every one of the bits represents whether the sampling decision was triggered by each of the seven sample sources or not, namely the connection table (first bit), the address table as source address (second bit), the address table as destination address (third bit), the port table as source port (fourth bit), the port table as destination port (fifth bit), the static filter sampling ratio (sixth bit), and/or the default sampling ratio (seventh bit)
UU	1	*Unused* This bit is unused, and should always be 1

Table B.4: Remapping of the 16 bit Ethernet Header's Protocol Field

Note that the current encoding does not permit processing IPv6 traffic.

[1]The rationale of associating the 00 code for invalid packet is to ensure that the normal codes for the ethertypes we support are invalid codes in our system. This way, a shim can detect easily that there is no device at the other side of the link, without waiting for a timeout after it sent the OPEN message.

[2]Note that a packet reaching the shim implies of course that it has been shunted. The MD field refers to the decision before sampling is taken into consideration. MD is therefore used to differentiate between two cases: (a) the device decision was really to shunt, and (b) the decision was to drop or to forward, but sampling implied a copy was sent to the engine. In the latter case, the packet has also been forwarded or dropped by the device, and therefore the shim must not try to reinject it back into the device. Note also that a packet may have been shunted and sampled at the same time. The shim knows the difference by checking the SA field.