University of Crete
Computer Science Department

Building Efficient Network Traffic Monitoring Systems Under Heavy Load

Antonis Papadogiannakis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in Computer Science
in the Graduate Division
of the University of Crete

March 2014
Heraklion, Greece

University of Crete
Computer Science Department

Building Efficient Network Traffic Monitoring Systems Under Heavy Load

Dissertation submitted by

Antonis Papadogiannakis

in partial fulfilment of the requirements for
the Ph.D. degree in Computer Science.

Author: _____
Antonis Papadogiannakis

Committee approvals: _____
Evangelos P. Markatos
Professor, University of Crete, Thesis Supervisor

_____
Constantine Dovrolis
Professor, Georgia Tech

_____
Sotiris Ioannidis
Principal Researcher, ICS-FORTH

_____
Angelos Bilas
Professor, University of Crete

_____
Apostolos Traganitis
Professor, University of Crete

_____
Kostas Magoutis
Assistant Professor, University of Ioannina

_____
Polyvios Pratikakis
Researcher, ICS-FORTH

Departmental approval: _____
Panos Trahanias
Professor, University of Crete, Chairman of the Department

Heraklion, March 2014

Building Efficient Network Traffic Monitoring Systems Under Heavy Load

Antonis Papadogiannakis

Abstract

Network traffic monitoring is the basis for a multitude of systems, such as intrusion detection, network forensics, and traffic classification systems, which support the robust, efficient, and secure operation of modern computer networks. However, building efficient network monitoring systems has become a challenging task. Emerging network monitoring applications become more demanding in terms of memory and CPU resources, due to the increasingly complex analysis operations they need to perform on the monitored traffic. Moreover, many network monitoring applications need to analyze the captured traffic at higher protocol layers. This need for reconstructing high-level entities results in increased application complexity and reduced performance. At the same time, the volume of traffic that should be analyzed in today's network links increases significantly. This leads to a growing demand for more resources to monitor the network traffic at line speeds, while it is very likely that the deployed monitoring systems will become overloaded. Even worse, attackers are able to intentionally overload a network monitoring system to impede its correct operation and pass malicious activities over the network undetected, as the existing systems do not provide protection against such attacks. Therefore, there is an increasing need for building efficient and robust network monitoring systems that will provide intelligent overload control mechanisms, will be able to defend against sophisticated attacks, and will utilize recent advances in the available commodity hardware.

In this dissertation we address the above issues, and we propose new techniques and frameworks to improve the performance, accuracy, and robustness of network monitoring systems when processing high volumes of traffic using commodity hardware. Our thesis is that we need to enrich the lower layers of a network monitoring system with intelligence based on flow-level information from the transport layer, in order to build efficient network monitoring systems under heavy load. First, we show that rearranging the captured packet stream based on source and destination port numbers can lead to significant performance benefits due to

improved memory access locality. We implement this technique, which we call as *locality buffering*, within a popular packet capture library, and we show its performance improvements in common network monitoring applications. To improve the accuracy of an overloaded Network-level Intrusion Detection System (NIDS), we suggest to focus on the first few bytes of each connection, a technique we call as *selective packet discarding*. Our evaluation shows that this approach can significantly improve the effectiveness of a NIDS under extreme load. To defend against overload attacks, we propose *selective packet paging*: a technique based on a two-layer memory management system to prevent packet loss, and on a randomized detection approach to find and isolate packets attacking the network monitoring system. To fill the semantic gap we identified between monitoring applications, which need to analyze network traffic at higher protocol layers, and monitoring libraries, which deliver just raw IP packets, we present the design, implementation, and evaluation of the *Stream capture library (Scap)*: a new multicore-aware framework for stream-oriented network traffic monitoring. Scap captures and delivers to user-level programs reassembled transport-layer streams, allowing for a wide variety of performance optimizations, such as hardware-assisted stream truncation, prioritized packet loss, and flexible stream reassembly. Finally, we show that our ideas can be applied in other problems of network monitoring systems as well, such as long-term network traffic recording and reducing the detection latency of an energy-efficient NIDS. To build more efficient and secure network monitoring systems, all these techniques we propose rely on the fact that monitoring applications are actually interested in a stream-oriented analysis.

Thesis Advisor: Professor Evangelos Markatos

Σχεδιασμός Αποδοτικών Συστημάτων για την Εποπτεία Δικτύων Υψηλού Φόρτου

Αντώνης Παπαδογιαννάκης

Περίληψη

Η εποπτεία της κίνησης ενός δικτύου αποτελεί την βάση για μια πληθώρα συστημάτων, όπως συστήματα ανίχνευσης επιθέσεων, μελέτης ηλεκτρονικών εγκλημάτων, και συστήματα κατηγοριοποίησης της κίνησης του δικτύου. Αυτά τα συστήματα συμβάλλουν σημαντικά στην αξιόπιστη, αποδοτική, και ασφαλή λειτουργία των σύγχρονων δικτύων. Όμως η δημιουργία αποδοτικών συστημάτων για εποπτεία δικτύων έχει γίνει ένα αρκετά δύσκολο έργο. Οι σύγχρονες εφαρμογές για εποπτεία δικτύων γίνονται πιο απαιτητικές σε πόρους συστημάτων όπως μνήμη και κύκλοι του επεξεργαστή, εξαιτίας της ολοένα και πιο περίπλοκης ανάλυσης που πρέπει να εφαρμόσουν στην κίνηση του δικτύου που παρακολουθούν. Επιπλέον, πολλές από αυτές τις εφαρμογές πρέπει να αναλύσουν την κίνηση του δικτύου σε πρωτόκολλα υψηλότερων επιπέδων. Αυτή η ανάγκη για την ανασύνθεση μηνυμάτων και οντοτήτων σε υψηλότερα επίπεδα κάνει τις εφαρμογές αυτές πολύ πιο περίπλοκες και μειώνει την απόδοσή τους. Την ίδια στιγμή, ο όγκος της κίνησης που θα πρέπει να αναλυθεί από αυτές τις εφαρμογές αυξάνει σημαντικά στα σημερινά δίκτυα. Αυτό οδηγεί σε μια αυξανόμενη ζήτηση για περισσότερους πόρους για την ανάλυση της κίνησης ενός πολύ γρήγορου δικτύου, ενώ είναι πολύ πιθανό ότι ένα τέτοιο σύστημα εποπτείας θα υπερφορτωθεί. Επιπλέον, ένας κακόβουλος χρήστης μπορεί να υπερφορτώσει σκόπιμα ένα σύστημα εποπτείας ενός δικτύου έτσι ώστε να εμποδίσει την σωστή λειτουργία του και να περάσει απαρατήρητες κακόβουλες δραστηριότητες μέσω αυτού του δικτύου, καθώς τα υπάρχοντα συστήματα δεν παρέχουν προστασία εναντίον τέτοιων επιθέσεων. Επομένως, υπάρχει μια αυξανόμενη ανάγκη για την σχεδίαση και υλοποίηση αποδοτικών και εύρωστων συστημάτων εποπτείας δικτύων τα οποία θα μπορούν να παρέχουν έξυπνους μηχανισμούς για περιπτώσεις υπερφόρτωσης, θα είναι σε θέση να αμυνθούν απέναντι σε εξελιγμένες επιθέσεις, και θα μπορούν να αξιοποιούν αποδοτικά τις δυνατότητες που προσφέρει το hardware.

ι

Σε αυτή τη διατριβή αντιμετωπίζουμε τα παραπάνω προβλήματα, και προτείνουμε νέες τεχνικές και συστήματα για την βελτίωση της απόδοσης, της ακρίβειας και της αξιοπιστίας συστημάτων εποπτείας δικτύων όταν αυτά επεξεργάζονται μεγάλο όγκο κυκλοφορίας χρησιμοποιώντας εξοπλισμό χαμηλού κόστους και ευρείας χρήσης. Η θέση μας είναι ότι για να φτιάξουμε αποδοτικά συστήματα εποπτείας δικτύων που έχουν πολύ μεγάλο φόρτο πρέπει να εμπλουτίσουμε τα χαμηλότερα επίπεδα του συστήματος με ευφυΐα που βασίζεται σε πληροφορίες από το επίπεδο μεταφοράς (transport layer). Αρχικά, δείχνουμε ότι η αναδιάταξη της ροής των πακέτων με βάση τους αριθμούς θύρας (port numbers) μπορεί να βελτιώσει σημαντικά την απόδοση λόγω της βελτιωμένης τοπικότητας στις προσπελάσεις μνήμης. Υλοποιήσαμε αυτήν την τεχνική, την οποία ονομάσαμε *locality buffering,* μέσα σε μια δημοφιλή βιβλιοθήκη για εποπτεία ενός δικτύου, και δείχνουμε την βελτίωση της απόδοσης που προσφέρει σε τυπικές εφαρμογές εποπτείας δικτύων. Για να βελτιώσουμε την ακρίβεια ενός συστήματος ανίχνευσης δικτυακών επιθέσεων που είναι υπερφορτωμένο, προτείνουμε να εστιάσουμε στα πρώτα bytes της κάθε σύνδεσης όταν το σύστημα έχει πολύ μεγάλο φόρτο, μια τεχνική που αποκαλούμε ως επιλεκτική απόρριψη πακέτων. Τα αποτελέσματά μας δείχνουν ότι αυτή η προσέγγιση μπορεί να βελτιώσει σημαντικά την αποτελεσματικότητα ενός συστήματος ανίχνευσης δικτυακών επιθέσεων σε περιπτώσεις πολύ μεγάλου όγκου δεδομένων. Για να αμυνθούμε ενάντια σε επιθέσεις που υπερφορτώνουν σκόπιμα το σύστημα, προτείνουμε μια νέα τεχνική που ονομάσαμε επιλεκτική σελιδοποίηση πακέτων. Αυτή η τεχνική βασίζεται σε ένα σύστημα διαχείρισης μνήμης δύο επιπέδων για να αποτρέψει την απώλεια πακέτων, και σε μια μέθοδο ανίχνευσης αυτών των επιθέσεων που χρησιμοποιεί τυχαιότητα ώστε να εντοπίσει και να απομονώσει τα πακέτα που επιτίθενται στο σύστημα. Επίσης, εντοπίσαμε ένα κενό μεταξύ του τι χρειάζονται οι εφαρμογές εποπτείας ενός δικτύου και του τι προσφέρουν τα συστήματα που υπάρχουν σήμερα: ενώ οι εφαρμογές πρέπει να αναλύσουν την κίνηση του δικτύου σε υψηλότερα πρωτόκολλα, οι υπάρχουσες βιβλιοθήκες παρέχουν απλά IP πακέτα. Για να καλύψουμε αυτό το κενό, παρουσιάζουμε τον σχεδιασμό, την υλοποίηση και την αξιολόγηση της βιβλιοθήκης *Scap (Stream capture library).* Η Scap παρέχει ένα καινούργιο framework με εγγενή υποστήριξη για συστήματα με πολλούς πυρήνες φτιαγμένο για την ανάλυση της κίνησης ενός δικτύου σε υψηλότερα πρωτόκολλα και βασισμένο στην προγραμματιστική αφαίρεση της ροής (stream). Έτσι, η Scap δίνει στις εφαρμογές την κίνηση ενός δικτύου σε ανακατασκευασμένα μηνύματα στο επίπεδο μεταφοράς (transport layer), αντί σε απλά IP πακέτα, επιτρέποντας έτσι πολλές βελτιώσεις στην απόδοση. Επίσης προσφέρει καινούργιες δυνατότητες, όπως περικοπή του μεγέθους μιας ροής με την βοήθεια της κάρτας δικτύου, ορισμός προτεραιοτήτων στις ροές και ευέλικτη ανασύνθεση των πακέτων σε μηνύματα υψηλότερων επιπέδων. Τέλος, δείχνουμε ότι οι ιδέες μας μπορούν να εφαρμοστούν και σε άλλα προβλήματα σχετικά με την εποπτεία δικτύων, όπως είναι η μακροχρόνια καταγραφή της κίνησης και η μείωση του χρόνου ανίχνευσης δικτυακών επιθέσεων σε ένα σύστημα με χαμηλή κατανάλωση ενέργειας. Οπότε, για να φτιάξουμε πιο

αποδοτικά και πιο ασφαλή συστήματα εποπτείας, όλες οι παραπάνω τεχνικές που προτείνουμε βασίζονται στο γεγονός ότι οι εφαρμογές ενδιαφέρονται τελικά να αναλύσουν την κίνηση του δικτύου με βάση τις συνδέσεις που υπάρχουν σε υψηλότερα πρωτόκολλα.

Επόπτης: Καθηγητής Ευάγγελος Μαρκατος

Acknowledgments

This thesis was performed at the Distributed Computing Systems (DCS) Laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH). The thesis was financially supported by ICS-FORTH and in part by the IST project LOBSTER, the FP7 project SysSec, and the GCC project funded by the European Union.

I am deeply grateful to my advisor, Professor Evangelos Markatos, for his valuable advice and continuous guidance during my studies, for his brilliant ideas and our long discussions, and for his significant contribution in this work. I am also grateful to Dr. Michalis Polychronakis for his constant support and for his contribution in this work. It was truly a valuable experience and I really enjoyed working with them.

I would also like to thank Dr. Sotiris Ioannidis and Professor Constantine Dovrolis for participating in my PhD advising committee and mainly for their valuable feedback during my PhD studies and my research work. I also thank Professor Angelos Bilas, Professor Apostolos Traganitis, Professor Kostas Magoutis, and Dr. Polyvios Pratikakis for participating in my PhD defense committee and for their useful feedback and comments on this thesis.

I deeply thank ICS-FORTH for providing all the equipment and all means needed to perform this work. It was a really nice and fun place to work and spend many hours of my life on the last few years! It also offered me the opportunity to work on numerous very interesting projects, and collaborate with brilliant researchers.

I would like to give special thanks to Christos Papachristos for his technical support and significant help to find and use the proper equipment for this work. He was always available and willing to help me. I would like to thank all the past and current members of the DCS laboratory for their collaboration, for making the lab a fun place to work, and for being good friends. Specifically, I thank Nikos Tsikudis, Giorgos Vasiliadis, Dr. Demetris Antoniades, and Michalis Orfanoudakis for contributing in works related to this thesis. I greatly appreciate their help. I also thank Dr. Sotiris Ioannidis, Laertis Loutsis, Dr. Vassilis Papaefstathiou, Panagiotis Papadopoulos, Apostolis Zarras, and Thanasis Petsas for working with me in other research projects and papers during my PhD. I would also like to thank Dr. Elias Athanasopoulos for useful discussions and feedback, as well as Lazaros Koromilas, Dr. Iasonas Polakis, Zacharias Tzermias, Giorgos Saloustros, Stamatis Volanis, Nick Nikiforakis, Manolis Stamatogiannakis, Meltini Christodoulaki, and all the

other past and current members of the DCS laboratory I did not mention by name. Many thanks go to Antonis Krithinakis and Panagiotis Garefalakis for their support and for having many fun times inside and outside of the lab.

Finally, I am really grateful to my parents, Charidimos and Aikaterinh, for their love and support, and I deeply thank Despoina for her constant support and encouragement all these years. I am also thankful to many friends and family members that are too many to mention by name.

Contents

List of Figures

List of Tables

1 Introduction

Ensuring the correct and secure operation of Internet applications continues to be a significant challenge. Along with the phenomenal growth of the Internet, the volume and complexity of Internet traffic is constantly increasing. Emerging highly distributed applications, such as media streaming, cloud computing, and peer-to-peer file sharing systems, demand for increased bandwidth and improved performance. Moreover, the number of mobile devices that are connected to the Internet increases tremendously every day. At the same time, the number of attacks against Internet-connected systems continues to grow at alarming rates. Besides the ever-increasing number and severity of security incidents, we have also been witnessing a constant increase in attack sophistication.

As networks grow larger and more complicated, with more applications deployed over the Internet, and as security incidents increase and become more sophisticated, effective network monitoring is becoming an essential operation for understanding, managing, and improving the performance and security of modern computer networks. For example, Network-level Intrusion Detection Systems (NIDS) inspect network traffic (both packet headers and payloads) to detect known attacks [113, 124], pinpoint compromised computers [61], and even identify previously unknown (i.e., zero-day) threats [116, 133]. Similarly, traffic classification tools inspect network packets (both headers and payloads) to identify different communication patterns and spot potentially undesirable traffic such as file sharing, unsolicited packets, and background radiation [1, 12, 79]. Traffic recording systems are used to store network packets for long-term periods to allow for network forensics analysis, data loss detection, and other types of retrospective analysis [62, 83, 89, 108]. Network monitoring applications are also being used to identify attack sources, to trace packet trajectories, to collect data that facilitate traffic engineering, and to find network operation parameters.

Therefore, network traffic monitoring is getting increasingly important for a large set of Internet users and service providers, such as ISPs, NRNs, computer and telecommunication scientists, security administrators, and managers of high-performance computing infrastructures. Installing monitoring and security systems at the network level has certain advantages compared to host-based installations: it is much easier to deploy, manage, and update a single monitoring or security middlebox for the whole network, instead of host-based monitoring or security systems at each client in a large network. Thus, there is a large market today focused on network monitoring, network surveillance, and network security. At the same time, there is a growing interest by researchers for this area, to improve the performance and accuracy of such systems, given the very dynamic nature of the network traffic, network applications, and cyber attacks.

Network monitoring is the capture and analysis of the network traffic of an organization. Typically, network monitoring systems are deployed close to the access links that connect an organization to Internet, so that they have a wide view of the organization's network traffic. The base of a network monitoring system is a *packet capture* subsystem, which is responsible to capture the traffic that passes through the monitored links and deliver it to the monitoring applications for traffic analysis. Such systems typically operate in two different modes: (i) in *passive* mode, where a separate copy of each packet is captured and analyzed in parallel with actual network's operation, or (ii) in *inline* mode, where traffic is analyzed while passing through the network. In the inline mode, the monitoring systems are able to interfere in network's operation and packet forwarding. For example, NIDS [113, 116, 124], traffic classification systems [1, 12, 79], and NetFlow export probes [3, 73] are passive monitoring systems, while Intrusion Prevention Systems (IPS) [30, 114, 139] and firewalls [100] operate inline.

In this dissertation, we study the field of network traffic monitoring: we identify the limitations of existing approaches, we show the necessity for building efficient network monitoring systems, we discuss the main challenges towards this goal, and we propose techniques to improve the state-of-the-art works in this area. We design and implement our proposed approaches within existing monitoring frameworks and tools that are widely used by many applications today. We also implement new frameworks for network traffic monitoring aiming to improve runtime performance and application development, in order to explore and evaluate the benefits of our approaches and make them available to the network monitoring community.

1.1 The Need for Efficient Network Traffic Monitoring

While network traffic monitoring was traditionally used for relatively simple network measurement and analysis applications, or just for gathering packet traces that are analyzed off-line, in recent years it has become vital for a wide class of more CPU and memory intensive applications, such as intrusion detection systems [113, 116, 124], accurate traffic categorization [1, 12, 79], and NetFlow export

probes [3, 73]. Many of these applications need to inspect both the headers and the whole payloads of the captured packets, a process widely known as *deep packet inspection* (DPI) [60]. For instance, measuring the distribution of traffic among different applications has become a difficult task. Many applications today use dynamically allocated ports, or operate above popular protocols like HTTP. Therefore, they cannot be simply identified based on a well known port number. Instead, protocol parsing and several other heuristics, such as searching for an application-specific string in the packets payload [1, 12, 79], are commonly used. Similarly, intrusion detection systems, such as Snort [124] and Bro [113], need to be able to inspect the payload of network packets in order to detect at real time malware and intrusion attempts. Threats are identified using attack "signatures" that are evaluated by advanced pattern matching algorithms, regular expression matching, and other types of complex analysis on the captured packets.

To make meaningful decisions, these monitoring applications usually need to analyze network traffic at the transport layer and above [110]. For instance, NIDSs reconstruct the transport-layer data streams to detect attack vectors spanning multiple packets, and perform traffic normalization to avoid evasion attacks [41,66,119]. Similarly, several traffic classification applications are also based on the processing of each transport-layer stream. However, the existing frameworks for building network monitoring applications provide just raw IP packets. This *gap* between what applications need and what current frameworks provide leads to increased code complexity, to increased development time, and, most importantly, to increased packet processing time due to the further operations needed to reassemble IP packets into higher level entities.

The complex analysis operations of such demanding applications incur an increased number of CPU cycles spent on the processing of every captured packet. Consequently, this reduces the overall processing throughput that the application can sustain without dropping incoming packets. At the same time, as the speed of modern network links and their traffic volume increase, there is a growing demand for more efficient packet processing using commodity hardware that will be able to keep up with higher traffic loads. Also, the increased processing time of each packet may lead to increased queuing delays and to an overall increase in the latency of the monitoring applications. To allow for almost real-time network traffic monitoring, and for timely automatic responses against malicious activities that are detected in the monitored network, a low latency should be also guaranteed by the monitoring systems.

Moreover, all these network monitoring applications have always been depended on an efficient and reliable underlying packet capture mechanism. However, such network traffic monitoring systems are now called to operate in an unpredictable and sometimes hostile environment where transient traffic and malicious attackers may easily overload them up to the point where they cease to function correctly [109, 135]. For instance, attackers may send crafted packets that exploit algorithmic complexity attacks to intentionally overload a NIDS [135]. Unfortunately, traditional packet capturing systems, have not been designed for such

hostile environments and do not gracefully handle overhead conditions. For example, when faced with overload conditions and full packet queues, most packet capturing systems start to discard all incoming packets for as long as the overload persists and until it resolves itself. Thus, malicious packets may pass through the overloaded network monitoring system undetected with high probability, equal to the percentage of packet loss.

To keep up with higher traffic loads, more complex analysis, overload periods, and overload attacks, network operators can buy and use more hardware resources for network monitoring purposes. However, this will significantly increase the cost and the energy consumption of these monitoring systems. Contrary, we would like to build efficient network traffic monitoring systems that will be able to utilize the recent advances in today's commodity hardware, such as modern features of Network Interface Cards (NICs) and multicore processor architectures. At the same time, we need to achieve high performance and robustness for these systems: no packet loss due to extreme processing or traffic load, low processing latency, graceful response to overload conditions, and resilience against attacks targeting these network monitoring systems.

1.2 Challenges and Problem Statement

There are many research works proposing techniques, algorithms, and system designs to improve network monitoring and network-level intrusion detection systems. Moreover, there are several open source and commercial solutions for network monitoring and network security. We review related works in Chapter 3. However, we see several shortcomings of the existing network monitoring libraries and tools, and we show the need for improvements along these directions. In summary, we consider the following main challenges for building efficient network traffic monitoring systems:

- **Challenge 1:** Traffic monitoring systems should be able to handle *more resource-intensive monitoring applications* that spend significantly more processing time on each captured packet, due to the more complex operations they need to perform.

- **Challenge 2:** Traffic monitoring systems should be able to handle *higher traffic throughput*, as the speed and volume of network links tend to increase rapidly.

- **Challenge 3:** Traffic monitoring systems should be able to operate even when *the system is overloaded* with a best effort approach. That is, when there are no available CPU cycles to process all the captured traffic at the desirable level, the monitoring system should have an overload control to focus on the most important tasks or traffic subsets, and minimize the impact of the overload on the overall application's performance and accuracy. We believe

that it is very difficult to ensure that such overload situations will never occur, even with very careful provisioning. Thus, as the traffic load increases and monitoring applications becoming more resource demanding, the core of a network monitoring system should have mechanisms for graceful responses to such overload conditions.

- **Challenge 4:** Traffic monitoring systems should be able to tolerate *evasion attempts*, such as algorithmic complexity or denial of service attacks trying to intentionally overload the system [33, 34, 109, 119, 135], or evasion attempts based on TCP segmentation [27, 41, 66, 150]. Thus, we need to develop mechanisms to detect and mitigate attacks against the monitoring systems, so that they will be able to operate correctly under the presence of adversaries and evasion attempts, even in case of unknown attacks and vulnerabilities that have not yet been discovered.

- **Challenge 5:** Traffic monitoring systems should have a *low latency* to allow for timely automatic reactions when malicious activities are detected. As monitoring applications operate at real time, a low latency is necessary to guarantee a responsive system for successful network management or attack prevention.

- **Challenge 6:** Many applications need to monitor network traffic at *higher protocol layers*, such as transport-layer streams or application-specific protocols. On the other hand, existing libraries for the development of monitoring applications provide just raw IP packets. This leads to increased complexity and runtime overhead for reassembling packets into higher-level entities.

- **Challenge 7:** Traffic recording systems need to *store a high traffic volume for a long period using limited storage* for retrospective analysis. Thus, they need to increase the retention period by reducing the storage needed, e.g., using effective compression techniques or by selectively storing the most useful traffic.

- **Challenge 8:** Traffic monitoring systems should use modern *commodity hardware* in order to be cost-effective and easy to deploy. Thus, they should utilize in the best possible way the recent advances in commodity hardware, such a features of modern NICs and multicore processor architectures.

- **Challenge 9:** Frameworks and systems that provide support for network monitoring applications should offer *performance optimizations that are transparent* from the programmers and network operators.

In this dissertation we propose and we study approaches that try to address all the above challenges. Given the necessity for efficient network traffic monitoring, we aim to design and build new systems and frameworks, or improve the existing ones, to facilitate the development of efficient network monitoring applications, by

improving their runtime performance and their security in a transparent way from the programmer. The proposed frameworks will run over commodity hardware, utilizing recent advances and features offered by the hardware vendors, and over general purpose operating systems that are typically used today for deploying such applications. Moreover, we want to design monitoring systems that will be able to operate under heavy processing and traffic load. They should be able to handle even the cases when the CPU cycles needed by a monitoring application to analyze the incoming traffic are more than the available cycles in the hardware used. This means that we need to add domain-specific intelligence within the monitoring libraries and systems to focus on the most interesting traffic, or perform the most critical processing, when the available CPU cycles are not enough to process all traffic at the same level.

1.3 Proposed Approaches

To address the above challenges, we aim to apply new techniques within the systems and frameworks that can be used by application developers for network traffic monitoring, and explore their efficiency and performance benefits. The techniques we propose rely on domain-specific knowledge of network monitoring applications. We believe that in order to make the lower layers of the monitoring systems more efficient under heavy load, we need to understand the needs of the applications at the higher layers. Thus, we add the proper functionality in the underlying libraries based on these observations to provide performance improvements and other features and optimizations.

In summary, in this dissertation we propose the following approaches:

- **Locality Buffering**: First, we explore how we can improve the memory access locality in network monitoring applications. To achieve that, we need to identify common memory access patterns in different monitoring applications and then generalize these patterns so that the underlying system can exploit them to improve data and code locality. We found that reordering the captured packet stream by clustering packets with the same port number before they are delivered to the application leads to improved code and data locality in a wide class of monitoring applications. This is because these applications keep state per each transport-layer stream or per each L7 application, while sorting packets based on port numbers tends to group together packets of the same stream and same L7 application. Improving the memory access locality results in significantly less cache misses, and thus to an overall performance improvement. We implemented this packet reordering technique, which we call as *locality buffering*, within the libpcap packet capture library [92]. This way, existing monitoring applications can transparently benefit from the reordered packet stream without code modifications.

- **Selective Packet Discarding**: Next, we explore ways to improve the behavior of a network monitoring system that is overloaded for a long time period. We use NIDS as a case study of a popular network monitoring application. To improve the accuracy of an overloaded NIDS, we propose to focus on the most important packets, i.e., the packets that are more important for attack detection. We designed, implemented, and evaluated *selective packet discarding (SPD)*: a technique based on the above idea. In our research we found that the first few packets of each connection, i.e., of each transport-layer stream, are more likely to contain an attack pattern. Also, these packets are more useful for the correct operation of the NIDS. Therefore, in cases of extreme load, the NIDS can benefit by proactively discard the less important packets, i.e., the packets towards the end of large streams in our case. This way, we can avoid random and uncontrolled packet loss by the underlying packet capture subsystem, part of which typically relies within the operating system's kernel.

- **Selective Packet Paging**: To tolerate attacks trying to overload a network monitoring application, like algorithmic complexity or other denial of service attacks, we propose *selective packet paging (SPP)*. Selective packet paging proposes (i) to add a second layer into memory management, so that excess packets will be stored to disk and will not be dropped when memory buffers are full, and (ii) to detect the packets that take too long to be processed, thus delaying the monitoring system, using a randomization-based detection approach, and push only these attack packets to secondary storage, while processing them with lower priority. We implemented this technique within libpcap, and we showed that it is able to make a network monitoring system resistant to any overload attack or any other overload situation.

- **Stream-oriented traffic capture and analysis:** We also identify a gap between existing monitoring libraries and applications' needs: although there is a need to monitor network traffic at the transport layer and beyond, existing libraries deliver only raw packets. To address this issue and fill this gap, we propose the *Stream capture library (Scap)*: the first network monitoring framework built from the ground up for stream-oriented traffic processing. Based on a kernel module that directly handles flow tracking and TCP stream reassembly, Scap delivers to user-level applications flow-level statistics and reassembled streams by minimizing data movement operations and discarding uninteresting traffic at early stages, while it inherently supports parallel processing on multicore architectures, and uses advanced capabilities of modern network cards.

- **Other Applications**: Finally, we show that we can use the same principles to develop techniques for improving two other problems in network monitoring systems as well. First, we focus on reducing the detection latency of an energy-efficient NIDS. This is necessary, as we found that the state-

of-the-art approaches for low-power design, such as frequency scaling and core deactivation, leads to a disproportionate increase in packet processing and queuing times, which has a negative impact on the detection latency and impedes a timely reaction of a NIDS to the incoming attacks. To address this issue, we present LEoNIDS: a NIDS architecture that provides *both* low power consumption *and* low detection latency at the same time, by identifying the packets that are more likely to carry an attack and giving them higher priority so as to achieve low attack detection latency. Then, we explore ways to store raw network traffic for long-term periods using constant storage, which is extremely beneficial for a multitude of monitoring and security applications. Towards this goal, we propose *RRDtrace*: a technique for storing full-payload packets for arbitrary long periods using fixed-size storage. RRDtrace divides time into intervals and retains a larger number of packets for most recent intervals. As traffic ages, an aging daemon is responsible for dynamically reducing its storage space by keeping smaller representative groups of packets using the proper sampling strategy.

1.4 Thesis and Contributions

Thesis statement: Building network traffic monitoring systems based on the transport-layer stream abstraction, instead of the IP packet abstraction, can make them resistant to overloads and improve their performance.

We intent to show that the above statement applies to modern network monitoring applications. Towards this goal, in this dissertation we make the following five main contributions:

- **Contribution 1:** We present *locality buffering*: a technique that can significantly improve the memory access locality in network monitoring applications by transparently reordering the captured packet stream based on port numbers. The improved code and data locality leads to less processing time per packet, and thus to increased processing throughput, due to the reduced number of CPU cache misses.

- **Contribution 2:** We show that the accuracy of an overloaded NIDS can be significantly improved by focusing on the most important packets for attack detection, which are the first few packets of each connection. Our approach, called *selective packet discarding*, monitors the system load and proactively discards packets towards the end of long flows based on a per-flow cutoff when the system becomes overloaded. This way, random and uncontrolled packet loss is avoided and the NIDS is able to detect almost all the attacks even under extreme load conditions.

- **Contribution 3:** To tolerate against overload attacks we propose *selective packet paging*: a technique based on (i) a two-layer memory management

system, which stores packet to secondary disk storage to ensure that no packet will be lost under overload, and (ii) a randomization-based detection approach, which finds and isolate any crafted packets that slowdown the monitoring system. We show that selective packet paging is able to tolerate any overload attack in a generic and effective way.

- **Contribution 4:** We identify a gap between monitoring applications and libraries: while applications are interested to analyze network traffic at transport layer and beyond, packet capture libraries provide just raw IP packets. This gap leads to increased application complexity and reduced performance. To address this issue, we introduce the *Stream capture library (Scap)*: the first network monitoring API build from stream-oriented traffic capture and analysis. Scap provides the user-level application with reassembled transport-layer streams, with inherent multicore support and a variety of features, such as subzero copy for stream truncation, prioritized packet loss for overload control, and flexible stream reassembly.

- **Contribution 5:** We show that our ideas can be applied to other problems of network monitoring systems, such as reducing the detection latency of an energy-efficient NIDS and store network traffic traces for long-term periods using fixed-size storage. To reduce the detection latency of a power-proportional NIDS, we identify the packets with higher probability to contain an attack and assign them higher priority to achieve fast detection. For long-term network traffic recording we choose to sample less traffic as traffic gets older. This way, we focus on most recent traffic while still keeping samples of past traffic. To select representative samples of older traffic, we explore different sampling strategies that fit to each monitoring application, such as packet sampling, flow sampling, or sampling the number of bytes we keep from the beginning of each flow.

1.5 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 provides some background information about network traffic monitoring systems and applications, and Chapter 3 presents related work in the broader area of network monitoring systems.

In Chapter 4 we demonstrate how the memory access locality of a network traffic monitoring application can be significantly improved by properly rearranging the packet stream, based on the source and destination port numbers. We present the design, implementation and evaluation of *locality buffering*: a technique that exploits this property to increase code and data memory access locality and significantly reduce L2 cache misses, resulting in an overall performance improvement in network monitoring systems.

In Chapter 5 we show that under extreme overload conditions, when packets will be unavoidably dropped by the system, a NIDS can improve its accuracy by

carefully selecting the packets that will be pro-actively dropped. To this end, we introduce *selective packet discarding*: an adaptive technique that pro-actively drops the less important packets for attack detection when an overload is identified by continuously monitoring the system's performance.

Chapter 6 introduces *selective packet paging*: an approach for tolerating overload attacks, such as algorithmic complexity attacks, against network monitoring systems. Selective packet paging combines a two-layer memory management system and a detection technique for packets aiming to slowdown the system. We present the design, implementation, and evaluation of selective packet paging under overload attacks.

In Chapter 7 we explain the need for efficient traffic capture and processing at the transport layer. To accommodate this need we propose the *Stream capture library (Scap)*: a stream-oriented framework for high-performance capturing and processing network traffic at the transport layer. We show that building a traffic processing framework using abstractions from transport layer allows for improved performance, reduced application development complexity, and many features, such as prioritized packet loss and subzero copy.

In Chapter 8 we explore how we can apply similar approaches in two other problems of network monitoring systems. First, we identify an energy-latency tradeoff for network-level intrusion detection systems. To resolve this tradeoff we propose to process the packets that are more likely to carry an attack with higher priority. We present the design, implementation and evaluation of *LEoNIDS*: a low-energy and low-latency NIDS, based on the above idea. Then, we study the problem of long-term network traffic recording with fixed-size storage. To allow the archiving of network traffic for long time periods we propose *traffic aging*, a mechanism that keeps more traffic for the most recent periods, and we implement this approach in a tool called *RRDtrace*. Also, we study how the traffic should be sampled as it gets older to match the requirements of retrospective network processing applications.

Finally, in Chapter 9 we summarize the contributions and results of this dissertation, and we outline research directions that can be explored in future work.

1.6 Publications

Parts of the work for this dissertation have been published in international refereed journals, conferences, and workshops:

- Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos. Scap: Stream-Oriented Network Traffic Capture and Analysis for High-Speed Networks. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC)*. October 2013, Barcelona, Spain.

- Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos. Tolerating Overload Attacks Against Packet Capturing Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. June 2012, Boston, MA, USA.

- Antonis Papadogiannakis, Giorgos Vasiliadis, Demetres Antoniades, Michalis Polychronakis, Evangelos P. Markatos. Improving the Performance of Passive Network Monitoring Applications with Memory Locality Enhancements. *Computer Communications*. Volume 35, Number 1, 2012.

- Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos. RRDtrace: Long-Term Raw Network Traffic Recording Using Fixed-Size Storage. In *Proceedings of the 18th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. August 2010, Miami, Florida.

- Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos. Improving the Accuracy of Network Intrusion Detection Systems Under Load Using Selective Packet Discarding. In *Proceedings of the European Workshop on System Security (EuroSec)*. April 2010, Paris, France.

- Antonis Papadogiannakis, Demetres Antoniades, Michalis Polychronakis, and Evangelos P. Markatos. Improving the Performance of Passive Network Monitoring Applications using Locality Buffering. In *Proceedings of 15th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. October 2007, Istanbul, Turkey.

Parts of this work have been also submitted for review:

- Antonis Papadogiannakis, Nikos Tsikudis, Evangelos P. Markatos. LEoNIDS: a Low-Latency and Energy-Efficient Network-Level Intrusion Detection System. **Under submission**.

- Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos. Tolerating Overload Attacks Against Deep Packet Inspection Systems. **Under submission** in *IEEE Journal on Selected Areas in Communications (JSAC)*

- 2014 Special Issue on Deep Packet Inspection: Algorithms, Hardware, and Applications.

- Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos. Stream-Oriented Network Traffic Capture and Analysis for High-Speed Networks. **Under submission** in *IEEE Journal on Selected Areas in Communications (JSAC) - 2014 Special Issue on Deep Packet Inspection: Algorithms, Hardware, and Applications*.

2
Background

In this chapter we give background information about network traffic monitoring systems and applications.

2.1 Network Traffic Monitoring Systems

Network monitoring applications analyze the network traffic by capturing and examining individual packets passing through the monitored link, which are then analyzed using various techniques, from simple flow-level accounting, to fine-grained operations like deep packet inspection. Network traffic monitoring systems are usually deployed close to the access link with which the monitored network is connected to the Internet. This way, the monitoring applications have a broad view of the network traffic. Depending on whether the monitoring systems operate on separate copies of each packet, or whether they operate within the path of the network, where packets are routed from source to destination, the network monitoring systems are divided into *passive* and *inline*.

Network monitoring applications run either in specialized hardware, or in general purpose commodity hardware. In this work we mostly consider the second case, although the same techniques we propose can be applied to monitoring systems running in specialized hardware as well, to address the same problems. Also, monitoring applications typical run over general purpose operating systems: they rely on a *packet capture* subsystem, part of which is usually implemented within the operating system kernel. As these operating systems are not optimized for network monitoring, their performance for packet capture is usually not optimum. This is because several data copies are required to transfer each captured packet from kernel to user space, along with an increased number of context switches

between kernel and user level processes, and increased processing time spent in kernel and interrupt handling. Therefore, several approaches have been proposed to reduce this overhead, which is unnecessary overhead for systems used specifically for network monitoring [58].

Applications are often interested in monitoring just a subset of the total packets passing through the network. Thus, *packet filtering* operation is performed to efficiently discard the uninteresting packets, usually within the operating system kernel, to avoid unnecessary packet copies to user level. Also, in several cases monitoring systems cannot capture and process all the packets of a network link. In this cases, packet sampling techniques are commonly used. However, such techniques are not adequate for all network monitoring and security applications. Another way to process a high traffic load is to distribute the packets among multiple CPU cores or multiple servers. In the rest of this section we give more information about packet capture systems, packet filtering, packet sampling, and distributed packet processing approaches.

2.1.1 Packet Capture Systems

The packet capture process is the journey of each packet from the wire until it is delivered to the passive monitoring application. First we briefly describe the regular packet capture process in Linux, and then we describe state-of-the-art techniques for improving the performance of packet capture.

Packet Capture in Linux

Most passive monitoring applications are built on top of libraries for generic packet capturing. The most widely used library for packet capturing is `libpcap` [92]. In Linux, libpcap is based on PF_PACKET socket for packet capture. Figure 2.1 depicts the whole process of packet reception in Linux. Packets travel from the Network Interface Card (NIC) through the kernel to reach the user level application. In Linux, this is achieved by issuing an interrupt for each packet or an interrupt for a batch of packets [98, 125]. Then, the kernel hands the packets over to every socket that matches the specified BPF filter [91]. In case that a socket buffer becomes full, the next incoming packets will be dropped from this socket. Thus, the size of the socket buffer affects the tolerance of a passive monitoring application in short-term traffic or processing bursts. Finally, each packet is copied to a memory mapped buffer that is accessible by the user-level application.

The packet capture mechanism consists of three tasks: (i) the Interrupt Service Routine (ISR) and (ii) the Soft Interrupt handler (Softirq) in kernel, and (iii) a user level task that calls a system call to access the next packet from kernel through a memory mapped buffer. There is a run time memory allocation from the interrupt handler for the received packet and a further copy to a memory mapped buffer, shared between kernel and user level. First, for each incoming packet the NIC issues an interrupt. The interrupt handler disables hardware interrupts, copies the

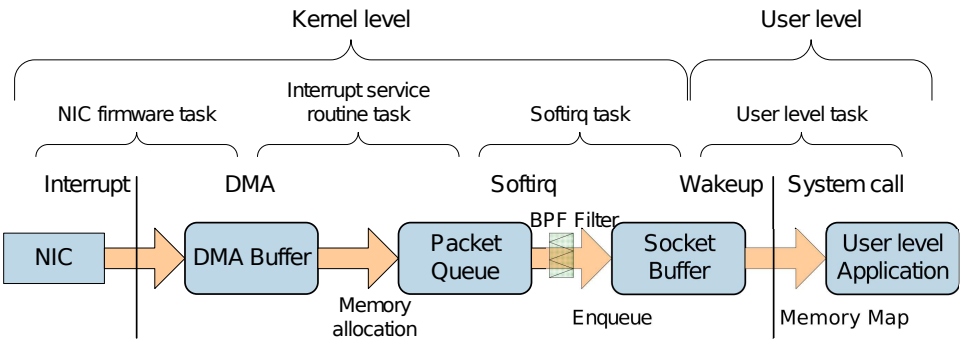


FIGURE 2.1: Packet capture architecture in Linux

packet from NIC to newly allocated memory, enqueues it to an incoming queue, issues a software interrupt, and then enables again interrupts. When the incoming queue becomes full (i.e., reaches the high congestion level) new incoming packets are being discarded from it, and thus from all the open sockets. Packets can also be dropped in NIC's buffer when interrupts are disabled.

When the control of the system is passed to kernel, the software interrupt handler runs. For PF_PACKET sockets, a software interrupt handler function is called for each packet and each open PF_PACKET socket. The software interrupt handler dequeues each packet from the incoming queue and calls a function for packet filtering for each open PF_PACKET socket. Packet filtering in Linux is very similar to BPF filtering [91]: if a filter has been assigned to an open PF_PACKET socket, each packet will be checked against it. If the packet does not match the filter, it will be discarded from this socket and will not be enqueued in the socket buffer queue. Else, if the packet matches the BPF filter, it is enqueued in the socket's receive queue. As the actual packet data does not change by any socket inside the kernel, they are not copied in each socket's buffer. Only the packet's metadata are copied in each buffer and a pointer is used to indicate the actual packet, which is shared by all PF_PACKET sockets that enqueue this packet. Before adding each packet to a socket's receive buffer, the software interrupt handler checks whether this queue if full. In case it is full, the packet is dropped from this socket's queue. The size of a socket's queue can be set by the applications. The finall step of the capture process is from `libpcap`. It reads each packet from linux kernel through an open PF_PACKET socket and a memory mapped buffer, shared between kernel and user level.

In summary, the main problems that limit the performance of packet capture in Linux are the following: (i) high interrupt service overhead per each packet, (ii) kernel-to-user-space context switching, (iii) data copy and memory allocation costs, and (iv) redundant protocol processing in kernel per each packet. Several techniques have been proposed to deal with one or more of the above issues, in order to improve the packet capturing process. We briefly mention some of them in the remaining of this section.

In order to avoid the kernel-to-user-space context switching and packet copies from kernel to user space for each captured packet, a memory-mapped ring buffer is used to store packets. The general principle of memory-mapping is to allow access from both kernel and user space to the same memory segment. The user lever applications are then able to read the packets directly from the ring buffer, thus avoiding a context switch to the kernel.

The ring buffer plays the same role as the socket buffer that we described earlier. The kernel is capable of inserting packets captured by the network interface into the ring buffer, while the user is able to read them directly from there. In order to prevent race conditions between the two different processes, an extra header is placed in front of each packet to ensure atomicity while reading and writing packets into the buffer. Whenever the processing of a packet is over, it is marked as read using this header, and the position in which the packet is stored is considered by the kernel as empty. The kernel uses an *end* pointer that points to the first available position to store the next arrived packet, while the user-level application uses a *start* pointer that points to the first non-read packet. These two pointers guarantee the proper operation of the circular buffer: The kernel simply iterates through the circular buffer, storing newly arrived packets on empty positions and blocks whenever the end pointer reaches the last empty position, while the user application processes every packet in sequence as long as there are available packets in the buffer.

However, in Linux packet capture, packets are still copied from DMA allocated memory to the memory mapped ring buffer by the software interrupt handler. *Zero copy* approaches can reduce data copy operations by sharing memory between different network stack layers within kernel and between kernel and user space.

To reduce interrupt load, some NICs issue a single interrupt for a group of packets instead for every packet received. To avoid interrupt overheads, polling based operation instead of interrupts has been proposed. Polling saves CPU time but increases the average receive latency. Moreover, periodically polling the device for incoming packets instead of interrupts is dangerous for loosing packets in a sudden burst or in high rates. Thus, the time period for polling is important in order to ensure that packets will not be lost in the NIC's queue. Furthermore, in case of low rates, polling very often the NIC without having any packers arrived may perform worse than interrupts. One solution is to try to guess the rate of incoming packets, based on previous measurements, and adapt the polling time period to this rate. Also, a common technique is to switch from interrupts to polling and backwards according to the incoming packet's rate. For low rates, interrupts are preferable while in high rates polling will lead to better performance. The NAPI polling driver [125] is available in Linux kernel. At high packet rates, the DMA buffer is polled to process the packets, thus interrupt service overhead and livelock phenomena are avoided.

Enhancing Packet Capture Performance

To improve the performance of packet capturing, several systems have been proposed to replace PF_PACKET: PF_RING [37, 55], netmap [123], PFQ [18], and PacketShader [65]. The common techniques used to overcome limitations of general purpose operating systems on packet capture are: (i) memory pre-allocation and re-use, (ii) parallel direct paths, (iii) memory mapping, (iv) zero-copy, (v) batch processing, (vi) CPU, interrupt, and memory affinity, and (vii) aggressive prefetching. These modifications are implemented in many different layers: NIC drivers, operating system kernel, and user-level libraries.

One of the state-of-the-art system that was designed to replace PF_PACKET in order to improve packet capture performance is PF_RING [37]. PF_RING focuses on bypassing packet copies and redundant protocol processing in Linux kernel for passive monitoring interfaces, reducing the packet's journey from the NIC to the user application. To achieve this, it uses a ring buffer that is memory mmaped in user space for storing the packets. Packets are copied by the NIC driver directly from the device to the ring buffer, which is accessible from user level processes. In case that a PF_RING socket is bounded to a network device, no further protocol processing will be done by the kernel, resulting in fewer CPU cycles spent per packet. This approach assumes that NIC is used only for passive capturing, and its packets cannot be used by any other socket expect PF_RING sockets.

Utilizing Multicore Processors

To utilize multicore processors, packet capture libraries rely on NICs to split the packets into multiple hardware queues. Many modern Ethernet cards support the partitioning of the incoming receive packet queue (RX-queue) into several RX-queues, one per each core. This modern hardware feature is called *Receive Side Scaling (RSS)* [75]. RSS uses a hash function on packet headers like IP addresses, port numbers, and protocol number to assign each packet into one of the different hardware queues. This way, the NIC achieves a good enough load balancing based on the hash function. Moreover, the traffic is balanced per-flow across the hardware queues: packets belonging to the same connection will be directed to the same queue, allowing for statefull inspection by accessing only a single RX-queue. To exploit parallelism offered by multicore architectures, the system typically uses a number of hardware queues equal to the number of the available CPU cores, and assigns one packet queue at each core.

To match this level of parallelism and speed-up the packet capture process in a multicore processor, the packet capture module in the operating system kernel need to spawn one thread per each core that polls a respective RX-queue [55]. For example, PF_RING kernel module exposes a different queue to user-level applications, to allow for multithreaded packet processing at user level.

Using Specialized Hardware

Another possible solution to accelerate packet capturing is to use specialized hardware optimized for high-speed packet capture. For instance, DAG monitoring cards [2] are capable of full packet capture at high speeds. Contrary to commodity network adapters, a DAG card is capable of retrieving and mapping network packets to user space through a *zero-copy* interface, which avoids costly interrupt processing. It can also stamp each packet with a high precision timestamp. A large static circular buffer, which is memory-mapped to user-space, is used to hold arriving packets and avoid costly packet copies. User applications can directly access this buffer without the invocation of the operating system kernel.

When using DAG cards, many performance problems occurred in Linux packet capturing can be eliminated, but at a price that is prohibitively high for many organizations. On the other hand, commodity hardware is always preferable and much easier to find and deploy for network monitoring. In addition, specialized hardware alone may not be enough for advanced monitoring tasks at high network speeds, e.g., intrusion detection.

2.1.2 Packet Filtering

Packet filtering is a commonly used technique for data reduction, that is necessary in monitoring systems due to limited processing and storage resources. Filtering is the deterministic selection of packets based on their content. Each packet captured by the monitoring network interface is matched against a set of rules in order to determine whether it is interesting for a particular application. Since the amount of traffic traveling on a network segment, especially in the network core, can be huge, filtering out irrelevant traffic is an essential step to reduce the demand in terms of storage and processing power on the monitoring and analysis tools.

Filtering usually implies extracting relevant fields from each packet and using their value to evaluate the rules. For example, packets from a given host or HTTP packets can be selected using filtering. Packet filtering is a special case of packet classification [64] that is used by various networking functions to group packets with common properties and separate packets that need to be processed differently.

Many approaches have been proposed and implemented for in-kernel packet filtering. High performance packet filtering is a very important issue in network monitoring, since it is usually the first task performed by a monitoring application in order to select the subset of the traffic it is interested in, and it has to be fast. Therefore, implementing packet filtering as early as possible is necessary for discarding the uninteresting packets at the first steps, avoiding further unnecessary processing of these packet by the kernel layers. Thus, implementations inside the OS kernel provides efficient packet filtering in this extent. Moreover, a packet filtering technique has to perform fast and efficient rule matching in the network packets, while it also has to be flexible and expressive enough to cover the applications needs in selecting the desirable portion of the traffic.

A popular in-kernel packet filter is BPF [91]. BPF defines an assembly-like language to perform comparisons between packet header fields and given values, and complex combinations using such comparisons that compose a tree-based expression evaluation. In Linux, one BPF filter can be assigned to each PF_PACKET socket. This is usually perform during the initialization process of a monitoring application. BPF filter evaluation in Linux is performed inside the kernel. Before the software interrupt routine enqueues each packet to a socket buffer, the packet is checked with the socket's filter – if a filter has been assigned to this socket. In order to evaluate the BPF filter, the kernel employs an optimized interpreter for the assembly-like BPF filter language. Thus, packets not matching the filter are immediately discarded before enqueued to the socket buffer and copied to user space. Packets that matched the filter are copied to user space for further processing by the applications. This makes the BPF filtering implementation efficient for improving the packet capturing performance, so applications should utilize filtering whenever it is possible to significantly improve their performance.

The packet filtering computation in case of BPF is fast due to the optimized interpreter for the assembly-like BPF language which describe the filter. BPF filtering language is expressive enough to support rules based on packet header fields for the basic network protocols. Thus, it can support filters based on IP addresses, port numbers, etc. However, BPF supports only stateless filtering (to keep its performance high). Also, filtering based on payload inspection is not possible. Generally, there is a tradeoff between flexibility and performance in packet filtering, so the proposed techniques either choose to focus in one of these two choices or try to keep a good balance between them.

Dynamic Packet Filtering

Dynamic packet filtering refer to on-line packet selection, where filtering criteria may frequently change over time. When a monitoring application cannot fully specify its criteria a priori, but the unknown part can only be determined at runtime, the filtering criteria have to be updated. For instance, filters that match the traffic of peer-to-peer applications, FTP transfers at dynamically generated port numbers, RTSP and other multimedia protocols, VoIP (SIP protocol), need to be updated with the dynamically negotiated port numbers. Also, NIDS may decide to perform more expensive and deep packet analysis on suspicious flows, thus upon the detection of a suspicious flow (at run time) a new or updated filter should be defined for this task.

With BPF and other techniques used for simple packet filtering operations, updating the filter is an expensive and slow operation, which may lead to a period that packets will de lost. This is because these filtering techniques were not designed for dynamic filtering. Proposed solutions for this problem either attempt to modify the simple packet filtering techniques, by reducing the cost of filter updating, or propose new dynamic filtering techniques [147, 152].

Hardware Filtering Capabilities at the NIC

Modern NICs offer advanced packet filtering capabilities in hardware. This way, packets can be filtered at the NIC layer, which is very efficient, as packets that do not match the specified filter will be discarded directly at the NIC. They, they will not be copied into the memory of the monitoring system, and no CPU cycle will be spent for uninteresting packets. Therefore, monitoring systems should utilize NIC's filtering capabilities and should push their filters into the hardware to significantly improve their performance. Dynamic packet filtering can be also implemented with hardware filters [38].

Moving Packet Processing to Kernel Packet Filter

Packet filtering techniques usually perform only packet discarding or packet copy operations in kernel, based on stateless rule matching. More complex packet processing is left for user level applications. Other approaches, such as xPF [76] and FFPF [19], move more packet processing capabilities from userspace into kernel, reducing context switches and improving the overall performance. For this purpose these techniques provide a richer programming framework for packet processing at the packet filter level. Also, they use in-kernel persistent memory to provide state in packet filtering.

2.1.3 Packet Sampling

Packet sampling is an other commonly used technique for data reduction, when the network traffic is too much to process or store. Sampling is the selection of a representative subset of packets. This subset is used to infer knowledge about the whole set of observed packets without processing them all. The selection may depend on packet's position, packet's content, or random decisions. While packet filtering is a deterministic selection of packets based on their content, which means that packets with the same properties will be always selected, packet sampling is a non-deterministic selection of packets, as it cannot be determined only from packet's content. The selection may be random or not: although it may also depend on some packet properties, it not necessary that all the packets matching these properties will be selected – only a percentage of them may be sampled.

Several different sampling strategies have been proposed [46]. They are currently being standardized by the Packet Sampling (PSAMP) Working Group of the Internet Engineering Task Forces (IETF) [71]. The strategy that each system selects for sampling is important for the accuracy of the monitoring application that will use the sampled data, since the set of sampled packets should be representative enough for the purpose needed. The deployment of a sampling strategy aims at the provisioning of specific characteristics of the parent population at a lower cost than a full census would demand. Therefore, in order to plan a suitable sampling strategy it is crucial to determine the needed type of metric that should be estimated and the desired degree of accuracy in advance. The metric of interest

can range from simple packet counts up to the estimation of whole distributions of flow characteristics.

Regarding the implementation of sampling techniques in a network monitoring architecture, similar with filtering it will be more effective when implemented as early as possible, e.g., within the OS kernel, in order to discard early the unsampled packets. PF_RING [37] implements sampling before storing the packets in the ring buffer, at the device driver level, while other techniques [59] implements packet sampling by extending the BPF implemenation.

Sampling is very often implemented in high-end routers today, for recording aggregated sampled traffic statistics like sampled NetFlows [51]. Packet sampling is an attractive techniques for routers because it is computationally efficient, as it requires minimal state and counters, and less storage. As routers' main operation is not traffic monitoring, they cannot offer much of their resources, such as CPU and memory, for this purpose. Moreover, the high bandwidth usage in the network to transport the collected data records from routers to a NetFlow collector machine can be significantly reduced using sampling. Adaptive filtering, based on the traffic load, has been also proposed.

Common Sampling Strategies

The sampling techniques can be classified into two main categories: *packet sampling* and *flow sampling*. Packet sampling is simple to implement with low CPU power and memory requirements. However, it is inaccurate for the inference of flow statistics such as the original flow size distribution. For instance, it is easy to miss the short flows. Flow sampling has been proposed as an alternative to overcome the limitations of packet sampling, e.g., to improve the accuracy in flow statistics inference. However, it imposes increased memory and CPU power requirements, which may be prohibitive for implementing in routers. To partially address this issue, especially to reduce memory and bandwidth requirements, techniques like smart sampling [48] and sample-and-hold [52] have been proposed as two variants of flow sampling with a focus on accurate estimation of heavy-hitters.

Systematic packet sampling involves the selection of packets according to a deterministic function. There are two ways to trigger the selection: count-based with the periodic selection every K packets or time-based, where a packet is selected every constant time interval. The first approach gives more accurate results in estimation of traffic parameters. Systematic sampling is easy to implement, but it is vulnerable to bias errors in metrics with related intervals and can be predicted by attackers when the data are used for security applications. In *random packet sampling*, a packet is selected with a probability basd on a random process. Contrary, *Random flow sampling* first classifies packets into flows based on flow construction rules. It then samples each flow with some probability. Using *random additive sampling*, the potential problems of systematic sampling are avoided. In this technique, the intervals between successive packet selections are independent random variables with a common distribution. In this way, synchronization and predictabil-

ity problems are avoided. In *n-out-of-N sampling*, n packets are randomly selected out of the total N packets. For this sampling scheme each packet has an equal chance of being drawn, and sample size is fixes. One way to achieve this random selection is to generate n different random numbers in the range of 1 to N. In *probabilistic sampling*, the decision whether a packet will be selected is made in accordance to a predefined selection probability. The sample size can vary for different trials. In *uniform probabilistic sampling*, each packet has the same selection probability, while in *non-uniform probabilistic sampling* the selection probability can vary for different packets. In the latter, the probability depends on the packet's content, e.g., to give higher probability to rare and important packets. Moreover, using *non-uniform probabilistic sampling* and *flow state*, packets can be selected based on the state of the flow they belong to or by the state of the other flows currently being monitored.

Sample-and-hold [52] performs a flow table lookup for each incoming packet to see if a flow entry for the packet's flow exists. If exists, the packet is selected and the flow entry is updated. Otherwise, if there is no flow entry for the packet, it is randomly selected and a new flow entry is created. The selection probability increases with the size of the packet. Unlike random packet sampling, all the subsequent packets of a flow are selected once the flow entry is created. *Smart sampling* [48] is a size dependent flow record selection algorithm that applies to complete flow records.

The sampling rate, i.e., the probability of packet selection in random sampling, directly affects the accuracy of the estimated metric. The more packets are selected, the better accuracy will succeed. On the other hand, high sampling rates lead to more resources consumption. Adaptive packet sampling techniques adjust the sampling rate to traffic load to further reduce memory consumption or to improve accuracy. Adaptive NetFLow [51] is based on this approach. In order to adjust properly the sampling rate, traffic rate prediction approaches are used. Rate constrained sampling approaches select a specified number of objects during a measurement interval, thus limiting the sampling rate. Reservoir sampling is a typical example. A special buffer (reservoir) holds a predefined number of samples. As more samples are being processed the contents of the reservoir may be replaced. Therefore, the contents of the reservoir each time represent a true random sample. Other approaches [49] work under strict resource constraints by sampling into a buffer of fixed size.

Applications of Packet Sampling

Packet and flow sampling has been used extensively to solve traffic engineering and pricing problems, such as heavy hitter identification [48, 52, 99, 120] and flow size estimation [122, 144]. For these applications, flow sampling provides better accuracy. To improve the estimation of TCP flow statistics using packet sampling, the use of TCP sequence numbers have been proposed [122]. The basic idea is that the presence of unsampled packets can be inferred by noting the increasing

byte counter given by the sequence number field of the TCP sampled packets. It is shown that this technique helps to fill the holes left by packet sampling in retaining information about the original flow sizes. However, it is still not as accurate as flow sampling. Other sampling approaches try to provide statistical performance similar to flow sampling with computational cost similar to packet sampling cost for TCP packets [144]. Such approaches also utilize TCP headers, such as sequence numbers and SYN/ACK/FIN flags.

While active measurements exchange probe packets between host pairs to measure network latency and packet loss rate, passive measurements exploit observations of traffic at two measurement points to infer the same metrics about network performance. For example, trajectory sampling [50] has been proposed as a method to correlate sampling of traffic at different locations. Routers sample packets only if a hash calculated over packet fields falls in a given set.

The use of sampled packet data has been also proposed for security analysis and anomaly detection. However, the impact of packet sampling on the accuracy of these applications depends on the sampling strategy and sampling rate [20, 88].

2.1.4 Load Shedding

Load shedding techniques reduce the load of a passive monitoring system when it is under severe stress, due to large traffic volumes or sudden traffic bursts. Such techniques should continuously monitor the system's performance for overloads, and upon the detection of an overload situation a subset of the incoming traffic will de discarding from processing. Their goal is to avoid uncontrolled packet loss, by selecting the traffic that will be lost, thus gracefully degrading their performance under excessive traffic load.

A load shedding approach [16] is proposed for the CoMo passive network monitoring infrastructure [70]. The CoMo monitoring system handles multiple arbitrary and continuous traffic queries. Passive monitoring applications are implemented with such traffic queries in CoMo, i.e., they define the subset of the traffic that their interested in, perform the desirable processing, and produce results. CoMo load shedding mechanism operates without explicit knowledge of the traffic queries. Instead, it extracts a set of features from the traffic streams to build an on-line prediction model of the query resource requirements. A feature is a counter that describes a specific property of a sequence of packets (e.g., number of unique source IP addresses). The features that best model the resource usage of each query are automatically identified and used to predict the overall load of the system. At the same time, measurements of the system resources are continuously performed, focused on CPU usage. By correlating the past counters of the selected features with their corresponding CPU usage measurements, the system can predict the CPU usage for the current counters of the same features. When an overload is predicted, the system applies load shedding techniques using uniform packet sampling and flow sampling. At the same time, it attempts to maintain the accuracy of the applications within acceptable levels.

Load shedding is also proposed as a defence to overload attacks in the Bro NIDS [113]. No specific packet discarding strategy is proposed, so the NIDS operator will be responsible to define one. This idea is based on the assumption that a NIDS operator can choose a packet discarding strategy that can be kept secret from attackers (security through obscurity).

2.1.5 Distributing the Load

An other way to deal with a highly loaded passive monitoring application, is to distribute its load in multiple CPUs. Multiple CPUs can be found by utilizing multiprocessing or multicore systems, as especially the latter have become commodity hadrware today. Libraries and techniques for efficient programming in multicore systems have been developed, and can be utilized by passive monitoring to exploit the offered parallelism. An other possible solution is to sustain a cluster of PCs to offload the workload from a single computer. In this case, a traffic splitter machine is usually responsible to capture and then split the packets to the cluster PCs, trying to equally balancing their load.

In both techniques the thoughtful selection of how the traffic will be split into the different CPUs, by the load balancer, is of significant importance. Traffic monitoring and analysis initially seems as a non-parallel task, because packets travel sequentially through the network links and they are captured in sequence by a single NIC. However, the large number of parallel transport-layer streams in the network, as we ll as the many different classes of packets and the different operations the applications perform on each class, offer several straightforward schemes for traffic splitting. For instance, flow-based traffic splitting strategy seems a good choice for most of the monitoring applications.

The packet capture process can be improved in multicore processors by exploiting the multiple RX queues in recent NICs and parallelize the packet capturing using all the available cores. Moreover, packets can be delivered efficiently to different ring buffers for utilizing multiple threads, so that the user-level packet processing can be parallelized using multiple threads or multiple processes.

However, today's commodity hardware and software is able to capture and process traffic up to few Gbit/sec rates, mainly due to limitations with the current PC buses bandwidth, CPU load and disk bandwidth, in case of writing packet traces to disk. To overcome this limitation, monitoring architectures for higher speeds have been proposed by distributing the traffic across a set of machines that are able to process lower speeds. To split the traffic in multiple machines, custom hardware has been designed. Also, to avoid the use of custom hardware modern features of Ethernet switches have been used to bundle lower speed interfaces into a single higher speed interface. Also, it is important to use the appropriate load balancing method for distributing equivalently the load to each link. Common switches support load balancing based on MAC addresses, IP addresses, port numbers and combinations. Load balancing based on flow identifiers (IP addresses and port numbers) is a good choice, in order to keep the corresponding data together.

Applications	Analysis
Traffic Recording	Full Packets, Storage
NetFlow Export	Packet Headers, Storage
Accounting and Billing	Packet Headers, Counters
QoS Monitoring	Packet Headers, Counters
Accurate Traffic Classification	Full Packets, Deep Packet Inspection
Intrusion Detection	Full Packets, Deep Packet Inspection
Anomaly Detection	Packet Headers, Machine Learning

TABLE 2.1: List with popular network traffic monitoring applications

2.2 Network Monitoring Applications

There is a multitude of network traffic monitoring applications, from security, forensics and network surveillance to network performance monitoring, accounting, and traffic classification. Table 2.1 lists some of the popular network monitoring applications today, along with the analysis each one require to perform on the captured traffic.

Packet capture and dump [142] is a very basic monitoring application, which simply captures packets from the wire and may save them to disk for retrospective offline analysis. It may be required to save full packets, i.e., both protocol headers and payload. In high traffic volumes, packet capture applications may focus on the elimination of packets that are of no interest, using packet filtering, to reduce CPU and memory usage, as well as disk requirements for the storage of large packet traces that can become very large in size in high traffic volumes.

Besides simple packet capture and dump, there are more advance network traffic recording systems that aim to store full packets for a relatively low period of time (retention) for retrospective analysis [83, 89, 108]. These systems are focus to increase the throughput that packets can be stored in modern storage systems, utilize the available store in the best possible way, e.g., by using compression, increase the retention period, and perform packet indexing operations to speed up retrospective queries on large volumes of captured traffic [56]. There are several potential applications of such systems: network forensics analysis, detection of compromised machines, matching of new signatures in past traffic, evaluation of new systems with real capture traffic, and any other types of retrospective analysis.

NetFlow [29] is one of the most commonly used protocol for monitoring network usage and collecting aggregated information about network traffic, such as NetFlow records. Collecting flow records is typically performed by routers, which gather and export such records to some collector. However, as the network speed increases, most routers are not able to do full flow analysis and have to use packet sampling to keep up [51]. Flow collection and export applications have been also moved to passive monitoring sensors running in commodity PCs, which can be used at high speeds when routers can not deliver flow records without using a very

low sampling rate. For efficient processing of high traffic volumes, flow export applications capture and process only the first few bytes of each packet, without loosing any valuable information, in order to capture just the protocol headers and to discard packet payloads. Such flow records can be used to show various information about the monitored network traffic.

Internet and application service providers use accounting applications to bill their customers based on their actual traffic or network usage. Thus, accounting applications are interested in accurately measuring various characteristics of network traffic like bandwidth usage, flow statistics and top bandwidth consuming hosts. These measurements require only header information per packet, while the use of *sampling* techniques has been proposed from several research works to deal with high network volumes with the minimum possible impact to the applications accuracy.

QoS and performance monitoring applications focus in estimating useful network metrics based on passive measurements, such as network Round-Trip Time [77], application-level Round-Trip Time [54] and throughput, packet retransmissions [50], packet reordering [95], one-way delay and jitter, and packet loss ratio [106]. Header only capturing is enough for all these applications, while some of them are based on flow record statistics [106]. For better resistance to high traffic speeds, *sampled* data are often used.

The accurate classification of network traffic among the layer-7 applications that generate the respective packets has become a difficult task, and requires more complex analysis. Instead of the traditional port-based classification, the use of dynamically allocated port numbers by many recent applications, and the use of popular protocols, like HTTP, to implement several different applications above it imposes more complex analysis in each packet, usually called as *deep packet inspection*). Therefore, protocol parsing, matching application-specific signatures within the packets' payload, and several other heuristics are commonly used [1, 12, 79]. This kind of processing is much more CPU intensive, which leads to lower processing throughput and higher packet drop rate. To improve the performance under high network load and reduce the CPU usage, traffic classification applications may select a subset of the total packets for inspection. For instance, only the first packet of each flow may be inspected for categorizing the flow [24], and only packets that does not belong to an already classified flow.

Network-level intrusion detection systems play an important role in the security of modern network architectures. A NIDS constantly monitors the network traffic in order to detect attacks or suspicious activity by matching packet data against known patterns [113, 124]. Such patterns, or rules, identify attacks by matching both header fields and payloads of the captured packets. While signature matching is a computationally intensive process [23], NIDS also need to perform operations like packet decoding, filtering, and IP/TCP stream reconstruction [66] to form a fully functional system. All these necessary operations make NIDS a CPU intensive application, significantly limiting its processing throughput [127]. Thus, under high network volumes a significant number of packets will be dropped, resulting

in undetected attacks. A resourceful attacker may intentionally overload a NIDS with a flood of crafted packets to passs real attack packets without inspection, as they may be dropped before inspected by the NIDS [119, 135].

Finally, anomaly detection approaches are based on profiling the network usage under normal behavior, raising alarms when abnormal conditions are detected. Packet headers or aggregated traffic statistics can be used for effective anomaly detection and behavior analysis using machine learning techniques, while the use of sampled data in anomaly detection metrics is still an open research question [20, 88].

2.2.1 Network-level Intrusion Detection Systems

Network-level Intrusion Detection Systems, such as Snort [124] and Bro [113], match the network packets against a set of rules, using multiple pattern matching techniques [9], regular expression matching [67], and other complex analysis. Moreover, they perform several other operations like protocol decoding, stream reassembly, and protocol normalization. These requirements for more complex per-packet inspection and the constant increase in network speeds have motivated numerous works for improving the performance of NIDSs. A lot of research works have been done to improve the algorithms used by a NIDS, e.g., the performance of pattern matching algorithms and regular expressions evaluation, to increase the overall throughput of NIDS. Other works propose the use of specialized hardware or splitting the load into multiple processing units. Such solutions indeed improve NIDS performance, but with an additional cost of a specialized hardware or multiple processors instead of a commodity PC.

The Snort NIDS

Snort [124] is a popular open source intrusion detection system based on *rules*, or *signatures*, which describe attacks and other suspicious activities. Using these rules it inspects the network traffic and whenever all options of one rule match against a network packet, it generates a respective alert at real time. Snort's rules are divided into two logical parts: the rule header and the rule options. The rule's header determines the rule action that will be triggered in a case of a match and also defines the protocol, the source and destination IP addresses, and the source and destination ports for the packets that will be checked against this rule. Thus, only a subset of the packet, specified in rule's header, will be matched against the rule's patterns and regular expressions. The rule's options contains a short attack description, the packet fields to be inspected, and the patterns or regular expressions that may lead to a full match. Snort parses all the given rules and represents them with automata and a two-dimension list.

The Snort receives packets through libpcap packet capture library. First, the packet is decoded up to the transport layer. Then, Snort can be configured with several preprocessors: each packet will pass from each of the enabled preproces-

sors before the inspection engine. Such preprocessors are responsible for defragmentation, protocol normalization, stream reassembly, while a user is also able to implement and enable custom preprocessors as well. Then, packets pass through the core detection engine, which is responsible for pattern matching operations and attack detection. Moreover, the user can add custom or existing detection plugins. Finally, there output plugins for alert generation and logging.

Using Specialized Hardware

To speed-up the inspection process, few NIDSs implementations are based on specialized hardware. Content addressable memory [156, 157] is suitable to perform parallel comparisons in packets payload against the NIDS rules, and significantly accelerate a NIDS. Many FPGA-based NIDS architectures have been implemented to accelerate pattern matching [13, 14], while network processors have been also used to speed-up NIDS operations [30, 35] Moreover, modern graphics processing units (GPUs) have been used to speed up pattern matching [149] for intrusion detection.

Distributing NIDS Load

To cope with high traffic volumes, NIDS architectures have been proposed to exploit multicore processors for parallel inspection [114], or to distribute the load across to multiple NIDS sensors [84, 128, 146]. A slicing mechanism divides the traffic into subsets, which are assigned to sensors in a way that each subset contains all the necessary evidence to detect a specific attack without any need for communication between the sensors. Moreover, the load balancer may use dynamic feedback from the sensors about their current load in order to adapt the traffic splitting accordingly and improve load balancing decisions.

NIDS Tuning and Performance Adaptation

Provisioning and tuning a NIDS is a significant process for its correct and effective operation. Besides a static NIDS configuration, some research approaches propose to dynamically reconfigure the NIDS based on the run-time conditions. By periodically measuring the NIDS's performance, the system may deactivate some less critical tasks and analysis [85], or may process a different subset of traffic that requires less processing from the NIDS [43]. Other research approaches try to predict the resource consumption of a NIDS based on a traffic sample, and use these predictions to automatically derive a suitable NIDS configuration [44].

2.2.2 Traffic Classification Systems

Peer-to-peer and multimedia applications are often violating corporate policies and when detected they are often blocked or rate limited. Consequently, such network applications have started to masquerade and obfuscate their traffic in order to avoid detection. They are not based on common ports any more and their protocols are becoming more complex, like the Skype's protocol for example, using payload encryption and other obfuscation techniques. Moreover, HTTP is becoming more and more popular for an increasing set of web-based applications. Thus, classifying the HTTP traffic into the actual applications that generate it requires more intelligent traffic analysis.

Accurate per-application traffic classification and identification techniques depend on *deep packet inspection*: both header and packet payload need to be inspected. They are usually based on application protocol specific patterns and signatures, common behaviors, and several other heuristics. This inspection requires more complex analysis and reduces the packet processing throughput. One way to improve the performance of traffic classification systems is to improve the signatures' accuracy and flow-based classification. Improving the accuracy of the signatures that will be matched against the network's traffic not only improve the classification accuracy, but it can also increase the throughput of such systems. This is because an accurate signature will limit the string search only to specific bytes within a packet or within a flow. Thus, the packet capture length may be limited to the first few bytes in many cases with carefully designed signatures. Such optimizations lead to significant performance improvements, as much less data is processed.

We refer to *flow-based classification techniques* as the techniques that try to classify transport-layer flows, according to the application that generates them. The flow-based classification techniques have a significant performance benefit when only the first few packets (or bytes) of a flow are inspected. Indeed, after inspecting the first few packets of a flow with no success, i.e., the flow cannot be classified, it is probably useless to continue searching for applications signatures. It is very likely that there is no signature for this type of traffic, and further processing will be only overhead to the system. Thus, it is preferable to mark this flow as "unknown". Similarly, when a packet matches an application signature, its flow is classified accordingly. Thus, the next packets of this flow will not be inspected, but will be only used for accounting the application's bandwidth usage. Therefore, flow-based traffic classification can significantly reduce the packet processing cost.

Other classification techniques perform less fine grained analysis, using only traffic and flow statistics, such as packet and flow sizes, interarrival times, and aggregated statistics, which are useful for traffic classification based on behavioral analysis and machine learning techniques [72, 78, 80].

2.2.3 Network Traffic Archiving Systems

To enable network forensics and other kinds of retrospective analysis, several traffic recording systems have been developed to store full packets in modern storage systems. Many of the proposed techniques focus on reducing the amount of traffic that should be stored for high-volume traffic links, either using efficient compression schemes [57, 141] or clever packet selection approaches [83, 89, 108]. Other systems focus on improving the throughput for writing network packets to storage systems, in order to support line rate [11, 40]. Finally, improving query responses in such systems have been studied, and packet indexing techniques have been proposed for this purpose [56].

3
Related Work

In this chapter we discuss related work in the broader area of network monitoring and network security systems.

3.1 Improving the Performance of Packet Capture

Braun et al. [21] and Schneider et al. [129] compare the performance of packet capturing libraries on different operating systems using the same hardware platforms and provide guidelines for system configuration to achieve optimal performance. Several research efforts [17,19,37,102,123] have focused on improving the performance of packet capturing through kernel and library modifications. These approaches reduce the time spent in kernel and the number of memory copies required for delivering each packet to the application.

Our proposed techniques can be combined with such optimizations to achieve even better performance. However, all these approaches operate at the network layer. Thus, monitoring applications that require transport-layer streams should implement stream reassembly, or use a separate user-level library, resulting in reduced performance and increased application complexity. In contrast, we propose Scap that operates at the transport layer and directly assembles incoming packets to streams in the kernel, offering the opportunity for a wide variety of performance optimizations and many features. Moreover, we show that memory access locality in passive network monitoring applications can be improved when reordering the packet stream based on source and destination port numbers [105, 111]. Thus, our locality buffering approach aims to improve the packet processing performance of the monitoring application itself, by exploiting the inherent locality of the in-memory workload of the application. Scap also improves memory access locality and cache usage in a similar manner when grouping packets into streams.

3.2 Taking Advantage of Multicore Systems

Previous work has dealt with how to use multicore systems to improve the performance of network monitoring systems. Fusco and Deri [55] utilize the receive-side scaling (RSS) feature of modern NICs [75], which split the network traffic in multiple RX queues, usually equal to the number of CPU cores, to parallelize packet capturing using all CPU cores. Moreover, packets are copied directly from each hardware queue to a corresponding ring buffer, which is exposed in user-level as a virtual network interface. Thus, applications can easily and efficiently split the load to multiple threads or processes without contention.

Sommer et al. [139] take advantage of multicore processors to pararrelize event-based network prevention systems, using multiple event queues that collect together semantically related events for in-order execution. Since the events are related, keeping them within a single queue localizes memory access to shared state by the same thread. Pesterev et al. [115] improve TCP connection locality in multicore servers using the flow director filters to optimally balance the TCP packets among the available cores.

3.3 Distributing The Traffic Load

Schneider et al. [130] show that commodity hardware and software is able to capture low traffic rates, mainly due to limitations with buses bandwidth and CPU load. To cope with this limitation, the authors propose a monitoring architecture for higher speed interfaces by splitting the traffic across a set of nodes with lower speed interfaces, using a feature of current Ethernet switches. The detailed configuration for load balancing is left for the application. Vallentin et al. [146] present a NIDS cluster based on commodity PCs. Some front-end nodes are responsible to distribute the traffic across the cluster's back-end nodes. Several traffic distribution schemes are discussed, focused on minimizing the communication between the sensors and keeping them simple enough to be implemented effectively in the front-end nodes. Hashing a flow identifier is proposed as the right choice.

3.4 Improving Memory Locality

The concept of locality buffering for improving passive network monitoring applications, and, in particular, intrusion detection and prevention systems, was first introduced by Xinidis et al. [153], as part of a load balancing traffic splitter for multiple network intrusion detection sensors that operate in parallel. In this work, the load balancer splits the traffic to multiple intrusion detection sensors, so that similar packets (e.g., packets destined to the same port) are processed by the same sensor. In this approach, however, the splitter uses a limited number of locality buffers and copies each packet to the appropriate buffer based on hashing on its

destination port number. Our locality buffering approach differs in two major aspects. First, we have implemented locality buffering within a packet capturing library, instead of a separate network element. To the best of our knowledge, our prototype implementation within the libpcap library is the first attempt for providing memory locality enhancements for accelerating packet processing in a generic and transparent way for existing passive monitoring applications. Second, the major improvement of our locality buffering approach is that packets are not actually copied into separate locality buffers. Instead, we maintain a separate index which allows for scaling the number of locality buffers up to 64K.

Locality enhancing techniques for improving server performance have been widely studied. For instance, Markatos et al. [90] present techniques for improving request locality on a Web cache, which results to significant improvements in the file system performance.

3.5 Packet Filtering

Kernel-level packet filtering improve the processing throughput of a monitoring application, as uninteresting packets are discarded in kernel and are never delivered in user level. A BPF filter [91] can be used for simple filtering needs, e.g., for choosing a subset of the traffic. Dynamic packet filtering reduces the cost of adding and removing filters at runtime [38, 147, 152]. Deri et al. [39] propose to use the NIC's flow director filters for common filtering needs. Besides from copying or discarding packets based on a stateless filter expression, other approaches, such as FFPF [19], xPF [76], and FLAME [10], allow applications to move simple tasks from user level to the kernel packet filter to improve performance. We suggests a relatively different approach in Scap: applications empowered with a Stream abstraction can communicate their stream-oriented filtering and processing needs to the underlying kernel module at runtime through the Scap API, to achieve lower complexity and better performance. For instance, Scap is able to filter packets within the kernel or at the NIC layer based on a flow size cutoff limit, allowing to set dynamically different cutoff values per-stream, while the existing packet filtering systems are not able to support a similar functionality.

3.6 TCP Stream Reassembly

Libnids [5] is a user-level library on top of libpcap for TCP stream reassembly based on the emulation of a Linux network stack. Similarly, the Stream5 [103] preprocessor, part of Snort NIDS [124], performs TCP stream reassembly at user level, emulating the network stacks of various operating systems. Scap shares similar goals with Libnids and Stream5. However, previous works treat TCP stream reassembly as a necessity [148], mostly for the avoidance of evasion attacks against intrusion detection systems [41, 66, 150]. On the contrary, Scap views transport-layer streams as the fundamental abstraction that is exported to network monitoring

applications, and as the right vehicle for the monitoring system to implement aggressive optimizations.

The main drawback of existing libraries for stream reassembly is their performance overhead, as they operate at user level, above the packet capturing subsystem. Scap provides the first OS subsystem for stream capturing that deliver reassembled transport-layer streams directly to user-level applications.

Handley et al. [66], Dharmapurikar and Paxson [41], and Vutukuru et al. [150] explain how an attacker can exploit protocol ambiguities to evade detection in a NIDS, and present the proper mechanisms for robust stream reassembly and normalization under the presence of adversaries. Scap performs follows these requirements in a strict stream reassembly mode, while it also supports a more relaxed *best-effort* stream reassembly mode that provides resiliency to packet drops due to system overloads.

3.7 Network-level Intrusion Detection Systems

The requirements for more complex per-packet inspection, the constant increase in network speeds, and the limited resources of commodity hardware have motivated numerous works for improving the performance of NIDSs. To speed-up the inspection process, many NIDS implementations use specialized hardware like content addressable memory [156, 157], FPGAs [13, 14], network processors [30, 35] and graphics processing units [149]. To cope with high traffic volumes, other approaches propose to distribute the load across multiple machines instead of using a single sensor [84, 128, 146], or to use multicore processors for parallel inspection [114]. These solutions offer almost linear processing throughput improvement, but with the additional cost of buying specialized hardware or processors with more cores. However, overloads are still possible in such systems in case of traffic bursts that exceed the NIDS processing throughput, or if one of the individual sensors of a NIDS cluster is overloaded. Furthermore, attackers may intentionally overload a NIDS to degrade its performance and increase their chances to evade detection [119, 135].

Lee et al. [85] propose to dynamically reconfigure the NIDS to provide optimal performance based on the current run-time conditions. By periodically measuring the performance of the system, the NIDS deactivates some less critical tasks and analysis, while an active firewall terminates offending connections. This approach focuses on determining the best configuration according to the given conditions and resource constraints. On the other hand, we propose to selectively discard packets with minimum impact to the detection accuracy without changing the NIDS configuration. We also propose selective packet paging to ensure that all packets will be inspected.

Another related approach by Dreger et al. [43] deals with packet drops due to overloads using load levels, which are precompiled sets of filters corresponding to subsets of traffic which the NIDS enables and disables depending on the workload.

Upon detecting an overload based on CPU measurements, the system backs off to a filter set that requires less processing capacity. The main difference of that approach from our selective packet discarding technique is that it relies on the NIDS operator to statically define an ordering of filters. On the other hand, selective packet discarding does not use any filters and does not require any knowledge for the network traffic or the processing times for specific types of traffic.

A more recent work from the same authors [44] presents a model for monitoring the resource usage of a NIDS, and then predicting its resource consumption. These predictions are used, based on a sample of the monitored traffic, to automatically derive a suitable NIDS configuration. While this approach can help the NIDS operator to find a suitable configuration automatically, we propose selective packet discarding to allow a NIDS to adapt its performance under high load even if no configuration can prevent overloads.

González and Paxson [59] present a technique that extends the NIDS with a secondary path for packet delivery in which the packets are randomly sampled, are not decoded, and TCP reassembly is not performed. While the packets in the main path are still processed as normal, the secondary path is tailored to different kind of analysis such as large connection and heavy hitters detection, and can be used to improve the performance for such kinds of analysis that do not require receiving all monitored packets.

3.8 Stream Truncation

The time Machine network traffic recording system [83] exploits the heavy-tailed nature of Internet traffic to reduce the number of packets that are stored on disk for retrospective analysis by applying a per-flow cutoff. Maier et al. [89] coupled Time Machine with a NIDS for enabling postmortem forensics queries. Our work [107] and Limmer and Dressler [86] use a per-flow cutoff under overload conditions, so that a NIDS focuses on the beginning of each connection and discards packets that are less likely to affect its detection accuracy. Canini, et al. [24] propose a similar scheme for traffic classification, by sampling more packets from the beginning of each flow. Lin et al. [87] present a system for storing and replaying network traffic, using an (N, M, P) scheme to reduce the traffic stored: they suggest to capture N bytes per flow and then M bytes per packet for the next P packets of the flow.

Scap shares a similar approach with the above works, but implements it within a general framework for fast and efficient network traffic monitoring, using the Stream abstraction to enable the implementation of performance improvements at the most appropriate level. For instance, Scap implements the per-flow cutoff inside the kernel or at the NIC layer, while previous approaches have to implement it in user space. As a result, they first receive *all* packets from kernel in user space, and then discard those that are not needed.

3.9 Load Shedding

Load shedding is proposed as a defence against overload attacks in the Bro NIDS [113]. However, the discarding strategy is not discussed, so the NIDS operator is responsible to define one. We propose selective packet discarding as a different load shedding technique, which suggests a new subset of traffic that should be discarded, based on the position of the packet within its corresponding flow [107]. Barlet-Ros et al. [16] also propose a load shedding technique in the CoMo passive monitoring infrastructure [70]. Using an on-line prediction model for the query resource requirements, the monitoring system sheds load under conditions of excessive traffic using uniform packet and flow sampling.

3.10 Packet Sampling

Packet sampling has been successfully applied for network flow monitoring in switches and routers to record aggregated sampled traffic statistics like sampled NetFlow [51], as the processing, memory, and storage resources in these devices are limited. Several sampling strategies have been proposed, which are currently being standardized by the Packet Sampling Working Group of IETF [71]. The choice of a suitable strategy depends on traffic characteristics or on statistics needed to be inferred.

Rate adaptive sampling has been proposed for dealing with traffic load variability. Adaptive NetFlow [51] uses traffic rate prediction techniques to adjust properly the sampling rate. Drobisz and Christensen [45] present an adaptive scheme based on CPU utilization and packet interarrival times. Choi et al. [28] determine the sampling probability adaptively according to traffic dynamics to accurate traffic load estimation. Hernandez et al. [68] use a predictive approach to anticipate load variations and adjust accordingly the sampling interval to meet sampling volume constraints. Similarly, rate constrained sampling approaches select a specified number of packets during a measurement interval. The method proposed by Duffield et al. [49] works under strict resource constraints by sampling into a buffer of fixed size. All these approaches adapt the sampling rate based on traffic load, while in RRDtrace we propose to adapt the sampling rate according to how old the stored traffic is, in order to provide better accuracy when processing the most recent traffic.

Brauckhoff et al. [20] examine the impact of packet sampling on anomaly detection metrics for the Blaster worm outbreak. Blaster uses random scanning in TCP port 135, so it can be detected using flow counters. However, flow counters are heavily affected from packet sampling. While packet and byte counters are not affected from sampling, they cannot detect Blaster anomalies. The flow entropy metric is shown to be more robust to packet sampling than flow counters. Mai et al. [88] examine the performance of volume and port scan anomaly detection methods with sampled data using four different strategies. The results show that all the

sampling strategies significantly degrade the performance of the detection algorithms. Among the four sampling schemes, random flow sampling introduces the least amount of distortion. Smart sampling [47] and sample-and-hold [52] are less resource intensive than random flow sampling, but perform poorly in the context of anomaly detection, since they miss small flows that are often related to attacks.

Overall, the impact of sampled data for anomaly detection metrics depends on the sampling strategy, sampling rate, and the analysis that should be performed. Applying packet sampling to signature-based NIDSs will result in missed attacks, while it is not appropriate for NIDSs that need to perform TCP stream reassembly. Contrary, we propose techniques like selective packet paging to ensure the inspection of all packets in cases where sampling or load shedding are not adequate.

3.11 Algorithmic Complexity Attacks

When deterministic finite automata (DFAs) are used for rule matching, each byte of traffic is examined exactly once, thus backtracking does not occur. However, DFAs experience exponential memory requirements and may not fit in memory in case of large rulesets. Nondeterministic finite automata (NFAs) reduce the memory requirements by allowing the matcher to be in multiple states concurrently. This is usually achieved through backtracking, which can be exploited for denial of service attacks. Several research works have proposed to improve the performance of regular expression matching [15, 136, 137, 154] by combining the benefits of both NFAs and DFAs. Kirrage et al. [82] present a static analysis technique to detect regular expressions that are vulnerable to algorithmic complexity attacks.

Cheng et al. [27] categorize and discuss a variety of evasion techniques against NIDSes. Smith et al. [135] introduce an algorithmic complexity evasion attack against the Snort NIDS, which exploits the backtracking behavior of rule matching in order to evaluate signatures at all possible string match offsets. They propose memoization as an algorithmic solution to deal with attacks targeting backtracking-based algorithms, by reducing the difference between the average and worst case costs. The idea is based on a memoization table that is used to store intermediate state that will not be recomputed.

Crosby and Wallach [34] present an algorithmic complexity attack that exploits deficiencies of common data structures, and propose new hashing techniques which sacrifice average case performance for worst case performance. Khan and Traore [81] propose a model to detect algorithmic complexity attacks based on historical information of execution time and input characteristics, using regression analysis. The authors propose to drop requests that do not conform with their model. However, in a NIDS this policy would result in successful evasion attacks, when an adversary sends an actual attack with a non conforming pattern. Afek et al. [8] propose the use of dedicated CPU cores to defend against algorithmic complexity attacks in NIDS.

Although the above works can address algorithmic complexity attacks against specific algorithms used in network monitoring systems, network monitoring and security software is still written so that it may be vulnerable to all sorts of algorithmic complexity attacks. Thus, monitoring and security applications still have to defend against complexity attacks that have not yet been seen in the wild, to ensure a robust and secure processing of the network traffic. Towards this direction, selective packet paging is a generic defense mechanism for any possible type of algorithmic attack or other overload situation.

3.12 Traffic Archiving Systems

A simple approach to increase retention when storing network traffic is to keep less data per packet. A common choice is to store only the first few bytes of each packet, which typically correspond to protocol headers. Solely from protocol headers, monitoring applications can infer useful information and network metrics, while this approach can reduce significantly the storage space [96] and thus increase retention. However, several monitoring applications, such as accurate traffic classification, as well as security applications commonly need to perform deep packet inspection operations, which require both the protocol headers as well as the payload of each packet [60]. Moreover, even with this significant reduction in storage requirements, retention time is still limited.

Another approach for efficient traffic recording is applied in the Time Machine system [83, 89], where only the first N bytes of each flow are recorded based on a per-flow cutoff. This approach leverages the heavy tailed distribution of flow sizes that is commonly found in Internet traffic, since most of the traffic in a high volume network comes from just a few flows. Therefore, most of the flows will not be affected by the cutoff and will be fully recorded, while recording only the beginning of a few large flows leads to significant savings in disk space. However, this technique cannot accurately estimate network metrics like total traffic volume and flow sizes. Furthermore, Time Machine stores approximately the same amount of traffic per day, and thus inevitably it can store traffic for a few days only. Then, it has to delete the past traffic.

Another solution that is commonly used to retain information about network usage in high volume networks for long-term periods is to maintain higher-level abstractions of the network traffic [25, 93, 121] or store aggregated data like NetFlow records [29]. Storing aggregated data instead of network packets can reduce dramatically the required disk space, while other higher-level abstractions can be used with fixed-size storage. However, such data formats limit significantly their usefulness. They can be adequate only for specific applications if the features of interest are known a priori. Any packet-level information will be lost, so many applications and deep packet inspection techniques do not work with aggregated traffic summaries. On the other hand, full-payload packet traces offer a rich source of information and allow for fine-grained analysis.

RRDtool [104] employs a Round Robin Database to store time-series data for very long periods in fixed-size storage using data aggregation. This feature of RRDtool has made it a popular choice for storing and visualising time-series data like temperatures, CPU load, and network metrics like bandwidth, delays, packet loss, and many other. RRDtool is based on an aging process using a consolidation function (usually average) to consolidate multiple primary data points to form a single consolidated data point. Therefore, older data will have less detail but will be representative for the corresponding time periods.

Cooke et al. [31] present a multi-format data storage technique that works with fixed storage and fixed time. First, packets are stored, and later on they are aggregated and transformed into flows as they age. Flows are finally aggregated into counters. Storage allocation algorithms divide the available storage between these different aggregation levels. The main shortcoming of this technique is that fine-grained analysis cannot be performed in old data, e.g., find possible undetected attacks or identify peer-to-peer and multimedia traffic using flow information. Moreover, having different data formats over time makes the analysis more difficult than having always the same data format.

Instead of storing actual packets, payload attribution techniques [118] store compressed digests of packet payloads. Based on an excerpt of a given packet payload, these techniques indicate the presence of packets that contained this exact payload and their source, destination and time of appearance on the network. Though, the actual payloads of the stored packets cannot be inferred. Such techniques are useful for forensics analysis and some security applications. Spring and Wetherall [140] present an algorithm for traffic compression by identifying and eliminating redundancy. Compression can effectively reduce the storage for protocols and applications with high redundancy.

The Bunker network tracing system [97] first writes all traffic to disk, to ensure that no packet is lost, and then performs offline the costly anonymization operations. This is similar to our proposed selective packet paging technique. However, our technique combines both memory and disk buffering in a hybrid architecture, so that packets are buffered in memory under normal situations to avoiding disk overheads, and packets are buffered to disk only when the memory buffer becomes full. While their system is oriented for anonymizing network traffic safely, our technique aims to protect network monitoring and security applications from denial of service attacks and other overloads.

Anderson et al. [11] present tools for recording packets at kernel-level to provide bulk storage at high rates. Hyperion [40] employs a write-optimized stream file system for high-speed storage using bloom filters to index stream data. Our work can utilize such techniques to improve performance for storing packets to disk.

Gigascope [32] is a stream database which offers an SQL-like language for queries on the packet stream, but does not focus on long-term archival. To speedup the query response times in packet traces, pcapIndex [56] proposes packet indexing techniques using compressed bitmaps.

3.13 Gap Analysis

Despite the significant research work that has been performed in this area, we identify several open research problems that are still needed to be addressed. First, the ever-increasing volume of network traffic and the increased complexity of monitoring applications require for improved performance. Although many of the previous works focus on performance, modern hardware offers new features and capabilities that can be used to improve the performance of network monitoring systems. As there are many network monitoring applications and tools developed, which comprise a large amount of code and lot of human effort, we would definitely prefer to achieve significant performance improvements in a completely transparent way from the applications, without any code modifications needed. Although some of the previous works try to offer backwards compatibility with legacy applications and libraries, we need more approaches that offer transparent optimizations and performance improvements.

We also identify a gap on overload control in network monitoring systems. Due to the high traffic load and application complexity, even after careful provisioning of a monitoring system, there is always a high probability that the system will get overloaded and start dropping packets from its memory buffers. Unfortunately, previous works have not focus on this problem. Contrary, we aim to fill this gap by providing the proper overload control, using domain-specific knowledge of the network monitoring applications.

Moreover, attackers are becoming more sophisticated and they try to pass malicious activities through the network without being detected by the monitoring systems. For example, attackers can intentionally overload a monitoring system to impede its detection capabilities, without significant effort, e.g, by exploiting algorithmic complexity vulnerabilities. Thus, we need to develop generic defenses against such evasion attempts. More specifically, a generic defense against any type of algorithmic complexity or overload attack is still missing.

Finally, we identify a gap between the current network monitoring frameworks and network monitoring applications. Although monitoring applications are interested in analyzing the network traffic at higher protocol layers, e.g., TCP streams, web pages, email messages, or SQL queries, the current monitoring frameworks offer just raw IP packets. These packets belong to multiple concurrent connections in the monitored network, and they can be out of order, retransmitted, and fragmented packets. Thus, applications need to reconstruct such packets into the higher-level entities to allow for useful processing, and perform protocol normalization to avoid evasion attempts based on TCP segmentation attacks. Unfortunately, however, this reconstruction leads to increased code complexity, increased application development time, and, most importantly, to reduced performance due to excessive data copies. We believe that building new network monitoring frameworks using the proper abstractions to fill this gap will facilitate the development of new monitoring applications, and will provide significant performance improvements and more opportunities for performance optimizations and useful features.

4

Enhancing Memory Access Locality

A common characteristic that is often found in network traffic monitoring applications is that they usually perform different operations on different types of packets. For example, a NIDS applies a certain subset of attack signatures on packets with destination port 80, i.e., it applies the web-attack signatures on packets destined to web servers, while a different set of signatures is applied on packet destined to database servers, and so on. Furthermore, NetFlow probes, traffic categorization, as well as TCP stream reassembly, which has become a mandatory function of modern NIDS [66], all need to maintain a large data structure that holds the active network flows found in the monitored traffic at any given time. Thus, for packets belonging to the same network flow, the process accesses the same part of the data structure that corresponds to the particular flow.

In all above cases, we can identify a *locality* of executed instructions and data references for packets of the same type. In this chapter, we present a novel technique for improving packet processing performance by taking advantage of this locality property which is commonly exhibited by many different passive monitoring applications. In practice, the captured packet stream is a mix of interleaved packets that correspond to hundreds or thousands of different packet types, depending on the monitored link. Our approach, called *locality buffering*, is based on reordering the packet stream that is delivered to the monitoring application in a way that enhances the locality of the application's code execution and data access, improving the overall packet processing performance.

We have implemented locality buffering in `libpcap` [92], the most widely used packet capturing library, which allows for improving the performance of a wide range of passive monitoring applications written on top of `libpcap` in a transparent way, without the need to modify them. Our implementation combines locality buffering with memory mapping, which optimizes the performance

of packet capturing by mapping the buffer in which packets are stored by the kernel into user level memory.

Our experimental evaluation using real-world applications and network traffic shows that locality buffering can significantly improve packet processing throughput and reduce the packet loss rate. For instance, the popular Snort IDS exhibits a 21% increase in the packet processing throughput and is able to process 67% higher traffic rates with no packet loss.

The rest of this chapter is organized as follows: In Section 4.1 we describe the overall approach of locality buffering, while in Section 4.2 we present in detail our implementation of locality buffering within the `libpcap` packet capturing library. Section 4.3 presents the experimental evaluation of our prototype implementation using three popular passive monitoring tools. Finally, Section 4.4 discusses limitations of our approach and future directions, and Section 4.5 summaries this chapter.

4.1 Locality Buffering

The starting point of our work is the observation that several widely used passive network monitoring applications, such as intrusion detection systems, perform almost identical operations for a certain class of packets. At the same time, different packet classes result to the execution of different code paths, and to data accesses to different memory locations. Such packet classes include the packets of a particular network flow, i.e., packets with the same protocol, source and destination IP addresses, and source and destination port numbers, or even wider classes such as all packets of the same application-level protocol, e.g., all HTTP, FTP, or BitTorrent packets.

Consider for example a NIDS like Snort [124]. Each arriving packet is first decoded according to its Layer 2–4 protocols, then it passes through several *preprocessors*, which perform various types of processing according to the packet type, and finally it is delivered to the main inspection engine, which checks the packet protocol headers and payload against a set of attack signatures. According to the packet type, different preprocessors may be triggered. For instance, IP packets go through the IP defragmentation preprocessor, which merges fragmented IP packets, TCP packets go through the TCP stream reassembly preprocessor, which reconstructs the bi-directional application level network stream, while HTTP packets go through the HTTP preprocessor, which decodes and normalizes HTTP protocol fields. Similarly, the inspection engine will check each packet only against a subset of the available attack signatures, according to its type. Thus, packets destined to a Web server will be checked against the subset of signatures tailored to Web attacks, FTP packets will be checked against FTP attack signatures, and so on.

When processing a newly arrived packet, the code of the corresponding preprocessors, the subset of applied signatures, and all other accessed data structures will be fetched into the CPU cache. Since packets of many different types will likely be highly interleaved in the monitored traffic mix, different data structures and code

FIGURE 4.1: The effect of locality buffering on the incoming packet stream.

will be constantly alternating in the cache, resulting to cache misses and reduced performance. The same effect occurs in other monitoring applications, such as NetFlow collectors or traffic classification applications, in which arriving packets are classified according to the network flow in which they belong to, which results to updates in a corresponding entry of a hash table. If many concurrent flows are active in the monitored link, their packets will arrive interleaved, and thus different portions of the hash table will be constantly being transferred in and out of the cache, resulting to poor performance.

The above observations motivated us to explore whether changing the order in which packets are delivered from the OS to the monitoring application improves packet processing performance. Specifically, we speculated that rearranging the captured traffic stream so that packets of the same class are delivered to the application in "batches" would improve the locality of code and data accesses, and thus reduce the overall cache miss ratio. This rearrangement can be conceptually achieved by buffering arriving packets into separate "buckets," one for each packet class, and dispatching each bucket at once, either whenever it gets full, or after some predefined timeout since the arrival of the first packet in the bucket. For instance, if we assume that packets with the same destination port number correspond to the same class, then interleaved packets destined to different network services will be rearranged so that packets destined to the same network service are delivered back-to-back to the monitoring application, as depicted in Figure 4.1.

Choosing the destination port number as a class identifier strikes a good balance between the number of required buckets and the achieved locality for commonly used network monitoring applications. Indeed, choosing a more fine-grained classification scheme, such as a combination of the destination IP address and port number, would require a tremendous amount of buckets, and would probably just add overhead, since most of the applications of interest to this work perform (5-tuple) flow-based classification. At the same time, packets destined to the same port usually correspond to the same application-level protocol, so they will trigger the same Snort signatures and preprocessors, or will belong to the same or "neighboring" entries in a network flow hash table.

However, sorting the packets by destination port only would completely separate the two directions of each bi-directional flow, i.e., client requests from server responses. This would increase significantly the distance between request and response packets, and in case of TCP flows, the distance between SYN and a SYN/ACK packets. For traffic processing operations that require to inspect both directions of a connection, this would add a significant delay, and eventually de-

crease memory locality, due to the separation of each bi-directional flow in two parts. Moreover, TCP reassembly would suffer from extreme buffering until the reception of pending ACK packets, or even discard the entire flow. For example, this could happen in case that ACKs are not received within a timeout period, or a packet is received before the SYN packet, i.e., before the TCP connection establishment. Furthermore, splitting the two directions of a flow would alter the order in which the packets are delivered to the application. This could cause problems to applications that expect the captured packets to be delivered with monotonically increasing timestamps.

Based on the above, we need a sorting scheme that will be able to keep the packets of both directions of a flow together, in the same order, and at the same time maintain the benefits of packet sorting based on destination port: good locality and lightweight implementation. Our choice is based on the observation that the server port number, which commonly characterizes the class of the flow, is usually lower than the client port number, which is usually a high port number randomly chosen by the OS. Also, both directions of a flow have the same pair of port numbers, in just reverse order. Packets in server-to-client direction have the server's port as source port number. Hence, in most cases, choosing the smaller port between the source and destination port numbers of each packet will give us the server's port in both directions. In case of known services, low ports are almost always used. In case of peer-to-peer traffic or other applications that may use high server-side port numbers, connections between peers are established using high ports only. However, sorting based on any of these two ports has the same effect to the locality of the application's memory accesses. Sorting always based on the smaller among the two port numbers ensures that packets from both directions will be clustered together, and their relative order will always be maintained. Thus, our choice is to sort the packets according to the smaller between the source and destination ports.

4.1.1 Feasibility Estimation

To get an estimation of the feasibility and the magnitude of improvement that locality buffering can offer, we performed a preliminary experiment whereby we sorted off-line the packets of a network trace based on the lowest between the source and destination port numbers, and fed it to a passive monitoring application. This corresponds to applying locality buffering using buckets of infinite size. Details about the trace and the experimental environment are discussed in Section 4.3. We ran Snort v2.9 [124] using both the sorted and the original trace, and measured the processing throughput (trace size divided by user time), L2 cache misses, and CPU cycles of the application. Snort was configured with all the default preprocessors enabled as specified in its default configuration file and used the latest official rule set [7] containing 19,009 rules. The Aho-Corasick algorithm was used for pattern matching [9]. The L2 cache misses and CPU clock cycles were measured using the PAPI library [6], which utilizes the hardware performance counters.

Performance metric	Original trace	Sorted trace	Improvement
Throughput (Mbit/s)	473.97	596.15	25.78%
Cache misses (per packet)	11.06	1.33	87.98%
CPU cycles (per packet)	31,418.91	24,657.98	21.52%

TABLE 4.1: Snort's performance using a sorted trace.

Table 4.1 summarizes the results of this experiment (each measurement was repeated 100 times, and we report the average values). We observe that sorting results to a significant improvement of more than 25% in Snort's packet processing throughput, L2 cache misses are reduced by more than 8 times, and 21% less CPU cycles are consumed.

From the above experiment, we see that there is a significant potential of improvement in packet processing throughput using locality buffering. However, in practice, rearranging the packets of a continuous packet stream can only be done in short intervals, since we cannot indefinitely wait to gather an arbitrarily large number of packets of the same class before delivering them to the monitoring application—the captured packets have to be eventually delivered to the application within a short time interval (in our implementation, in the orders of milliseconds). Note that slightly relaxing the in-order delivery of the captured packets results to a delay between capturing the packet, and actually delivering it to the monitoring application. However, such a sub-second delay does not actually affect the correct operation of the monitoring applications that we consider in this work (delivering an alert or reporting a flow record a few milliseconds later is totally acceptable). Furthermore, packet timestamps are computed *before* locality buffering, and are not altered in any way, so any inter-packet time dependencies remain intact.

4.2 Implementation within `libpcap`

We have chosen to implement locality buffering within `libpcap`, the most widely used packet capturing library, which is the basis for a multitude of passive monitoring applications. Typically, applications read the captured packets through a call such as `pcap_next` or `pcap_loop`, one at a time, in the same order as they arrive to the network interface. By incorporating locality buffering within `libpcap`, monitoring applications continue to operate as before, taking advantage of locality buffering in a transparent way, without the need to alter their code or link them with extra libraries. Indeed, the only difference is that consecutive calls to `pcap_next` or similar functions will most of the time return packets with the same destination or source port number, depending on the availability and the time constraints, instead of highly interleaved packets with different port numbers.

4.2.1 Periodic Packet Stream Sorting

In `libpcap`, whenever the application attempts to read a new packet, e.g., through a call to `pcap_next`, the library reads a packet from kernel and delivers it to the application. Using `pcap_loop`, the application registers a callback function for packet processing that is called once per each captured packet read from kernel by `libpcap`. In case that memory mapping is not supported, the packet is copied through a `recv` call from kernel space to user space in a small buffer equal to the maximum packet size, and then `pcap_next` returns a pointer to the beginning of the new packet or the callback function registered by `pcap_loop` is called. With memory mapping, the next packet stored by kernel in the shared ring buffer is returned to application or processed by the callback function. If no packets are stored, `poll` is called to wait for the next packet reception.

So far, we have conceptually described locality buffering as a set of buckets, with packets having the same source or destination port ending up into the same bucket. One straightforward implementation of this approach would be to actually maintain a separate buffer for each bucket, and copy each arriving packet to its corresponding buffer. However, this has the drawback that an extra copy is required for storing each packet to the corresponding bucket, right after it has been fetched from the kernel.

In order to avoid additional packet copy operations, which incur significant overhead, we have chosen an alternative approach. We distinguish between two different phases: the packet *gathering* phase, and the packet *delivery* phase. In the case without memory mapping, we have modified the single-packet-sized buffer of `libpcap` to hold a large number of packets instead of just one. During the packet gathering phase, newly arrived packets are written sequentially into the buffer by increasing the buffer offset in the `recv` call until the buffer is full or a certain timeout has expired. For `libpcap` implementation with memory mapping support, the shared buffer is split into two parts. The first part of the buffer is used for gathering packets in the gathering phase, and the second part for delivering packets based on the imposed sorting. The gathering phase lasts either till the buffer used for packet gathering gets full or till a timeout period expires.

Instead of arranging the packets into different buckets, which requires an extra copy operation for each packet, we maintain an index structure that specifies the order in which the packets in the buffer will be delivered to the application during the delivering phase, as illustrated in Figure 4.2. The index consists of a table with 64K entries, one for each port number. Each entry in the table points to the beginning of a linked list that holds references to all packets within the buffer with the particular port number. In the packet delivery phase, the packets are delivered to the application ordered according to their smaller port by traversing each list sequentially, starting from the first non-empty port number entry. In this way we achieve the desired packet sorting, while, at the same time, all packets remain in place, at the initial memory location in which they were written, avoiding extra costly copy operations. In the following, we discuss the two phases in more detail.

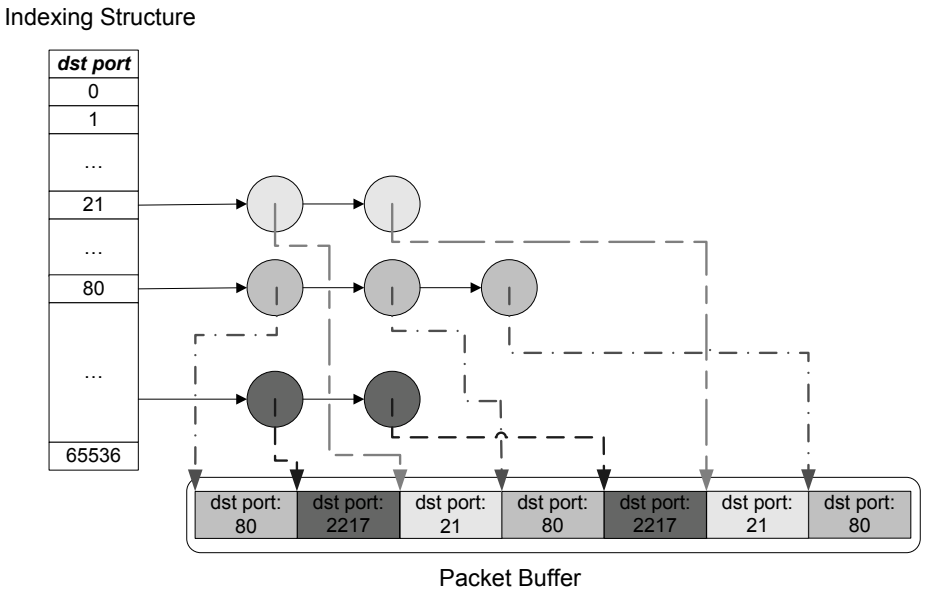


FIGURE 4.2: Using an indexing table with a linked list for each port, the packets are delivered to the application sorted by their smaller port number.

In the beginning of each packet gathering phase the indexing table is zeroed using `memset()`. For each arriving packet, we perform a simple protocol decoding for determining whether it is a TCP or UDP packet, and consequently extract its source and destination port numbers. Then, a new reference for the packet is added to the corresponding linked list. For non-TCP or non-UDP packets, a reference is added into a separate list. The information that we keep for every packet in each node of the linked lists includes the packet's length, the precise timestamp of the time when the packet was captured, and a pointer to the actual packet data in the buffer.

Instead of dynamically allocating memory for new nodes in the linked lists, which would be an overkill, we pre-allocate a large enough number of spare nodes, equal to the maximum number of packets that can be stored in the buffer. Whenever a new reference has to be added in a linked list, a spare node is picked. Also, for fast insertion of new nodes at the end of the list, we keep a table with 64K pointers to the tail of each list.

The overhead of this indexing process is negligible. We measured it using a simple analysis application, which just receives packets in user space and then discards them, resulting to less than 6% overhead for any traffic rate. This is because most of the CPU time in this application is spent for capturing packets and delivering them to user space. The overhead of finding the port numbers and adding a node to our data structure for each packet is negligible compared to packet capturing and other per-packet overheads. The overhead of making zero (through a

`memset()` call) the indexing table is also negligible, since we make it once for a large group of packets. In this measurement we used a simple analysis application which does not benefit from improved cache memory locality. For real world applications this overhead is even smaller, and as we observe in our experimental evaluation (Section 4.3) the benefits from memory locality enhancements outreach by far this overhead.

The system continues to gather packets until the buffer becomes full or a certain timeout has elapsed. The timeout ensures that if packets arrive with a low rate, the application will not wait too long for receiving the next batch of packets. The buffer size and the timeout are two significant parameters of our approach, since they influence the number of sorted packets that can be delivered to the application in each batch. Both timeout and buffer size can be defined by the application. Depending on the per-packet processing complexity of each application, the buffer size determines the benefit in its performance. In Section 4.3 we examine the effect that the number of packets in each batch has on the overall performance using three different passive monitoring applications. The timeout parameter is mostly related to the network's workload.

Upon the end of the packet gathering phase, packets can be delivered to the application following the order imposed by the indexing structure. For that purpose, we keep a pointer to the list node of the most recently delivered packet. Starting from the beginning of the index table, whenever the application requests a new packet, e.g., through `pcap_next`, we return the packet pointed either by the next node in the list, or, if we have reached the end of the list, by the first node of the next non-empty list. The latter happens when all the packets of the same port have been delivered (i.e., the bucket has been emptied), so conceptually the system continues with the next non-empty group.

4.2.2 Using a Separate Thread for Packet Gathering

In case that memory mapping is not supported in the system, a single buffer will be used for both packet gathering and delivery. A drawback of the above implementation is that during the packet gathering phase, the CPU remains idle most of the time, since no packets are delivered to the application for processing in the meanwhile. Reversely, during the processing of the packets that were captured in the previous packet gathering period, no packets are stored in the buffer. In case that the kernel's socket buffer is small and the processing time for the current batch of packets is increased, it is possible that a significant number of packets may get lost by the application in case of high traffic load.

Although in practice this effect does not degrade performance when short timeouts are used, we can improve further the performance of locality buffering in this case by employing a separate thread for the packet gathering phase, combined with the usage of two buffers instead of a single one. The separate packet gathering thread receives the packets from the kernel and stores them to the *write buffer*, and also updates its index. In parallel, the application receives packets for processing

from the main thread of libpcap, which returns the already sorted packets of the second *read buffer*. Each buffer has its own indexing table.

Upon the completion of both the packet gathering phase, i.e., after the timeout expires or when the write buffer becomes full, and the parallel packet delivery phase, the two buffers are swapped. The write buffer, which now is full of packets, turns to a read buffer, while the now empty read buffer becomes a write buffer. The whole swapping process is as simple as swapping two pointers, while semaphore operations ensure the thread-safe exchange of the two buffers.

4.2.3 Combine Locality Buffering and Memory Mapping

A step beyond is to combine locality buffering with memory mapping to further increase the performance of each individual technique. While memory mapping improves the performance of packet capturing, locality buffering aims to improve the performance of the user application that processes the captured packets.

The buffer where the network packets are stored in libpcap with memory mapping support is accessible from both the kernel and libpcap library. The packets are stored sequentially into this buffer by the kernel as they arrive, while the libpcap library allows a monitoring application to process them by returning a pointer to the next packet through pcap_next or calling the callback function registered through pcap_loop for each packet that arrives. In case the buffer is empty, libpcap blocks, calling poll, waiting for new packets to arrive. Packet processing is performed by a user-defined handler function that is registered through pcap_loop or pcap_dispatch and is called once for each packet that arrives.

After finishing with the processing of each packet, through the callback function or when the next pcap_next is called, libpcap marks the packet as read so that the kernel can later overwrite the packet with a new one. Otherwise, if a packet is marked as unread, the kernel is not allowed to copy a new packet into this position of the buffer. In this way, any possible data corruption that could happen by the parallel execution of the two processes (kernel and monitoring application) is avoided.

The implementation of locality buffering in the memory mapped version of libpcap does not require to maintain a separate buffer for sorting the arriving packets, since we have direct access to the shared memory mapped buffer in which they are stored. To deliver the packets sorted based on the source or destination port number to the application, we process a small portion of the shared buffer each time as a batch: instead of executing the handler function every time a new packet is pushed into the buffer, we wait until a certain amount of packets has been gathered or a certain amount of time has been elapsed. The batch of packets is then ordered based on the smaller of source and destination port numbers.

The sorting of the packets is performed as described in Section 4.2.1. The same indexing structure, as depicted in Figure 4.2, was built to support the sorting. The structure contains pointers directly to the packets on the shared buffer. Then, the

handler function is applied iteratively on each indexed packet based on the order imposed by the indexing structure. After the completion of the handler function, the packet is marked for deletion as before in order to avoid any race conditions between the kernel process and the user-level library.

A possible weakness of not using an extra buffer, as described in Section 4.2.2, is that if the batch of the packets is large in comparison to the shared buffer, a significant number of packets may get lost during the sorting phase in case of high traffic load. However, as discussed in Section 4.3, the fraction of the packets that we need to sort is very small compared to the size of the shared buffer. Therefore, it does not affect the insertion of new packets in the meanwhile.

In case of memory mapping, a separate thread for the packet gathering phase is not required. New incoming packets are captured and stored into the shared buffer by the kernel in parallel with the packet delivery and processing phase, since kernel and user level application (including the `libpcap` library) are two different processes. Packets that have been previously stored in buffer by kernel are sorted in batches during the gathering phase and then each sorted batch of packets are delivered one-by-one to the application for further processing.

4.3 Experimental Evaluation

4.3.1 Experimental Environment

Our experimental environment consists of two PCs interconnected through a 10 Gbit switch. The first PC is used for traffic generation, which is achieved by replaying real network traffic traces at different rates using `tcpreplay` [145]. The traffic generation PC is equipped with two dual-core Intel Xeon 2.66 GHz CPU with 4 MB L2 cache, 6 GB RAM, and a 10 Gbit network interface (SMC 10G adapter with XFP). This setup allowed us to replay traffic traces with speeds up to 2 Gbit/s. Achieving larger speeds was not possible using large network traces because usually the trace could not be effectively cached in main memory.

By rewriting the source and destination MAC addresses in all packets, the generated traffic is sent to the second PC, the passive monitoring sensor, which captures the traffic and processes it using different monitoring applications. The passive monitoring sensor is equipped with two quad-core Intel Xeon 2.00 GHz CPUs with 6 MB L2 cache, 6 GB RAM, and a 10 Gbit network interface (SMC 10G adapter with XFP). The size of memory mapped buffer was set to 60,000 frames in all cases, in order to minimize packet drops due to short packet bursts. Indeed, we observe that when packets are dropped by kernel, in higher traffic rates, the CPU utilization in the passive monitoring sensor is always 100%. Thus, in our experiments, packets are lost due to the high CPU load. Both PCs run 64bit Ubuntu Linux (kernel version 2.6.32).

For the evaluation we use an anonymized one-hour long trace captured at the access link that connects an educational network with thousands of hosts to the

Internet. The trace contains 58,714,906 packets, corresponding to 1,493,032 different flows, totalling more than 40 GB in size. To achieve high speeds, up tp 2 Gbit/s, we split the trace into a few smaller parts, which can be effectively cached in the 6 GB main memory, and we replay each part of the trace for 10 times in each experiment.

We measure the performance of the monitoring applications on top of the original version of `libpcap-1.1.1` and our mofidied version with locality buffering. The latter combines locality buffering with the memory mapping. For each setting, we measure the L2 cache misses and the CPU clock cycles by reading the CPU performance counters through the PAPI library [6]. Another important metric we measure is the percentage of packets being dropped by `libpcap`, which is occurred when replaying traffic in high rates due to high CPU utilization.

Traffic generation begins after the application has been initiated. The application is terminated immediately after capturing the last packet of the replayed trace. All measurements were repeated 10 times and we report the average values. We focus mostly on the discussion of our experiments using Snort IDS, which is the most resource-intensive among the tested applications. However, we also briefly report on our experiences with Appmon and Fprobe monitoring applications.

4.3.2 Snort

As in the experiments of Section 4.1.1, we ran Snort v2.9 using its default configuration, in which all the default preprocessors were enabled, and we used the latest official rule set [7] containing 19,009 rules. Initially, we examine the effect that the size of the buffer in which the packets are sorted has on the overall application performance. We vary the size of the buffer from 100 to 32,000 packets while replaying the network trace at a constant rate of 250 Mbit/s. We send traffic at several rates, but we first present results from constant 250 Mbit/s since no packets were dropped at this rate, to examine the effect of buffer size on CPU cycles spent and L2 cache misses when no packets are lost. We do not use any timeout in these experiments for packet gathering. As long as we send traffic at constant rate, the buffer size determines how long the packet gathering phase will last. Respectively, a timeout value corresponds to a specific buffer size.

Figures 4.3 and 4.4 show the per-packet CPU cycles and L2 cache misses respectively when Snort processes the replayed traffic using the original and modified versions of `libpcap`. Both `libpcap` versions use the memory mapping support, with the same size for the shared packet buffer (60,000 frames) for fairness. Figure 4.5 presents the percentage of the packets that are being dropped by Snort when replaying the traffic at 2 Gbit/s, for each different version of `libpcap`.

We observe that increasing the size of the buffer results to fewer cache misses, fewer clock cycles, less dropped packets, and generally to an overall performance improvement for the locality buffering implementations. This is because using a larger packet buffer offers better possibilities for effective packet sorting, and thus to better memory locality. However, increasing the size from 8,000 to 32,000

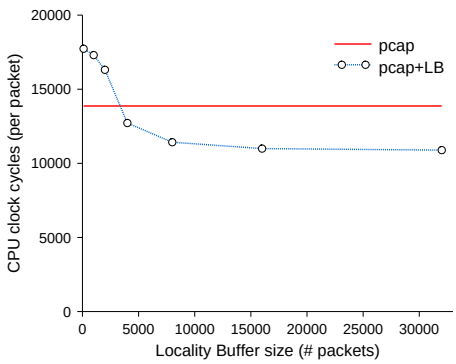


FIGURE 4.3: Snort's CPU cycles as a function of the buffer size for 250 Mbit/s traffic.

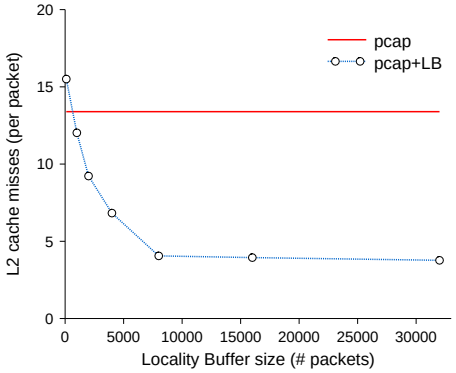


FIGURE 4.4: Snort's L2 cache misses as a function of the buffer size for 250 Mbit/s traffic.

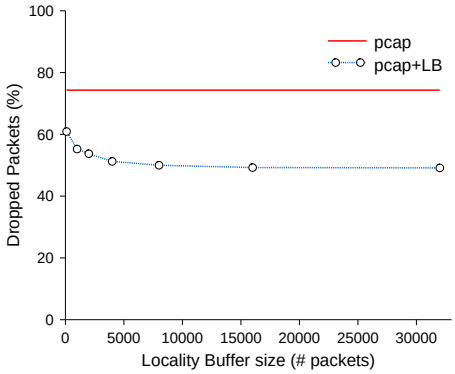


FIGURE 4.5: Snort's packet loss ratio as a function of the buffer size for 2 Gbit/s traffic.

packets gives only a slight improvement. Based on this result, we consider 8,000 packets as an optimum buffer size in our experiments. When sending traffic in a constant rate of 250 Mbit/s, with no timeout specified, 8,000 packets as buffer size roughly correspond to an 128 millisecond period at average.

We can also notice that using locality buffering we achieve a significant reduction in L2 cache misses from 13.4 per packet to 4.1, when using a 8,000-packet buffer, which is an improvement of 3.3 times against Snort with the original `libpcap` library. Therefore, Snort's user time and clock cycles are significantly reduced using locality buffering, making it faster by more than 20%. Moreover, when replaying the traffic at 2 Gbit/s, the packet loss ratio is reduced by 33%. Thus, Snort with locality buffering and memory mapped `libpcap` performs significantly better than using the original `libpcap` with memory mapping support. When replaying the trace at low traffic rates, with no packet loss, Snort outputs the

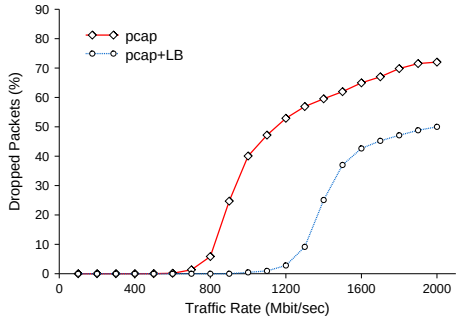


FIGURE 4.6: Packet loss ratio of the passive monitoring sensor when running Snort, as a function of the traffic speed for an 8,000-packet locality buffer.

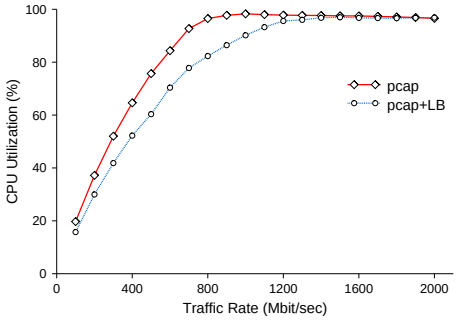


FIGURE 4.7: CPU utilization in the passive monitoring sensor when running Snort, as a function of the traffic speed for an 8,000-packet locality buffer.

same set of alerts with and without locality buffering, so the packet reordering does not affect the correct operation of Snort's detection process.

We repeated the experiment by replaying the trace in different rates ranging from 100 to 2,000 Mbit/s and in every case we observed a similar behavior. In all rates, 8,000 packets was found to be the optimum buffer size. Using this buffer size, locality buffering results in all rates to a significant reduction in Snort's cache misses and CPU cycles, similar to the improvement observed for 250 Mbit/s traffic against the original `libpcap`. The optimum buffer size depends mainly on the nature of traffic in the monitored network and on the network monitoring application's processing.

An important metric for evaluating the performance of our implementations is the percentage of the packets that are being dropped in high traffic rates by the kernel due to high CPU load, and the maximum processing throughput that Snort can sustain without dropping packets. In Figure 4.6 we plot the average percentage of packets that are being lost while replaying the trace with speeds ranging from 100 to 2,000 Mbit/s, with a step of 100 Mbit/s. The 2,000 Mbit/s limitation is due to caching the trace file parts from disk to main memory in the traffic generator machine, in order to generate real network traffic. We used a 8,000-packet locality buffer, which was found to be the optimal size for Snort when replaying our trace file at any rate.

Using the unmodified `libpcap` with memory mapping, Snort cannot process all packets in rates higher than 600 Mbit/s, so a significant percentage of packets is being lost. On the other hand, using locality buffering the packet processing time is accelerated and the system is able to process more packets in the same time interval. As shown in Figure 4.6, using locality buffering Snort becomes much more resistant in packet loss, and starts to lose packets at 1 Gbit/s instead of 600

Mbit/s. Moreover, at 2 Gbit/s, our implementation drops 33% less packets than the original `libpcap`.

Figure 4.7 shows Snort's CPU utilization as a function of the traffic rate, for rates varying from 100 Mbit/s to 2 Gbit/s, with a 100 Mbit/s step, and a locality buffer size of 8,000 packets. We observe that for low speeds, locality buffering reduces the number of CPU cycles spent for packet processing due to the improved memory locality. For instance, for 500 Mbit/s, Snort's CPU utilization with locality buffering is reduced by 20%. The CPU utilization when Snort does not use locality buffering exceeds 90% for rates higher than 600 Mbit/s, and reaches up to 98.3% for 1 Gbit/s. On the other hand, using locality buffering, Snort's CPU utilization exceeds 90% for rates higher than 1 Gbit/s, and reaches about 97% for 1.5 Gbit/s rate. We also observe that packet loss events occur due to high CPU utilization, when it approaches 100%. Without locality buffering, 92.7% CPU utilization for 700 Mbit/s results to 1.4% packet loss rate, while 98% utilization for 1.1 Gbit/s results to 47.2% packet loss.

In Figure 4.8 we plot the average percentage of dropped packets while replaying traffic with normal timing behavior, instead of sending traffic with a constant rate. We replay the traffic of our trace based on it's normal traffic patterns when the trace was captured, using the multiplier option of tcpreplay [145] tool. Thus, we are able to replay the trace at the speed that it was recorded, which is 88 Mbit/s on average, or at a multiple of this speed. In this experiment we examine the performance of Snort using the original and our modified `libpcap` in case of normal traffic patterns with packet bursts, instead of constant traffic rates. We send the trace using multiples from 1 up to 16, and we plot the percentage of dropped packets as a function of this multiplication factor. We use 8,000 packets as buffer size and 100 ms timeout.

We observe that locality buffering reduces the percentage of dropped packets in higher traffic rates, when using larger multiplication factors. Snort with locality buffering starts dropping packets when sending traffic eight times faster than the actual speed of the trace, while Snort with the original `libpcap` drops packets from four times faster speed. When replaying traffic 16 times faster than the recorded speed, which results to 1,408 Mbit/s on average, Snort with locality buffering drops 34% less packets.

Since both versions of `libpcap` use a ring buffer for storing packets with the same size, they are resistant to packet bursts at a similar factor. However, `libpcap` with locality buffering is faster, due the improved memory locality, and so it is more resistant to packet drops in cases of overloads and traffic bursts.

4.3.3 Appmon

`Appmon` [12] is a passive network monitoring application for accurate per-application traffic identification and categorization. It uses deep-packet inspection and packet filtering for attributing flows to the applications that generate them. We ran Appmon on top of our modified version of `libpcap` to examine its performance using

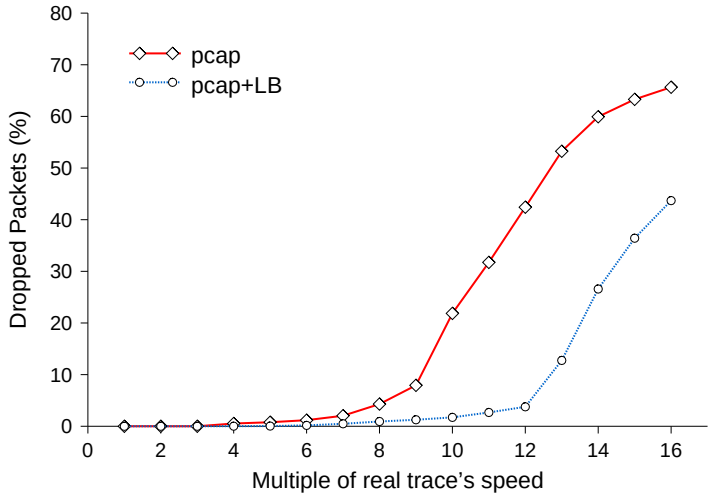


FIGURE 4.8: Packet loss ratio of the passive monitoring sensor when running Snort, as a function of the actual trace's speed multiple, with an 8,000-packet locality buffer and 100 ms timeout.

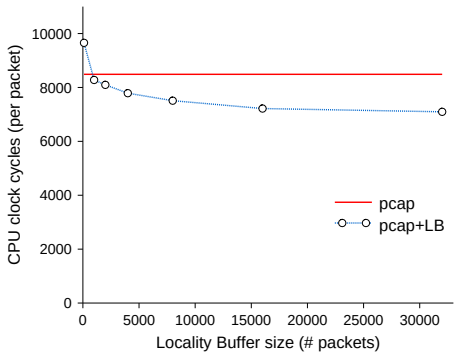


FIGURE 4.9: Appmon's CPU cycles as a function of the buffer size for 500 Mbit/s traffic.

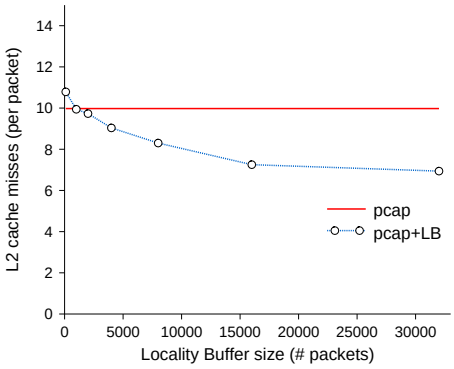


FIGURE 4.10: Appmon's L2 cache misses as a function of the buffer size for 500 Mbit/s traffic.

different buffer sizes that vary from 100 to 32,000 packets, and compare with the original `libpcap`. Figure 4.9 presents the Appmon's CPU cycles and Figure 4.10 the L2 cache misses measured while replaying the trace at a constant rate of 500 Mbit/s. At this rate no packet loss was occurred.

The results show that Appmon's performance can be improved using the locality buffering implementation, reducing the CPU cycles by about 16% compared to Appmon using the original `libpcap`. Cache misses are reduced by up to 31% for 32,000 packets buffer size and 28% for 16,000 packets. We notice that in case of Appmon the optimum buffer size is around 16,000 packets, while in Snort 8,000 packets size is enough to optimize the performance. This happens because App-

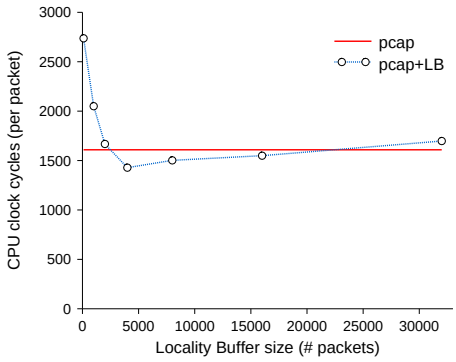


FIGURE 4.11: Fprobe's CPU cycles as a function of the buffer size for 500 Mbit/s traffic.

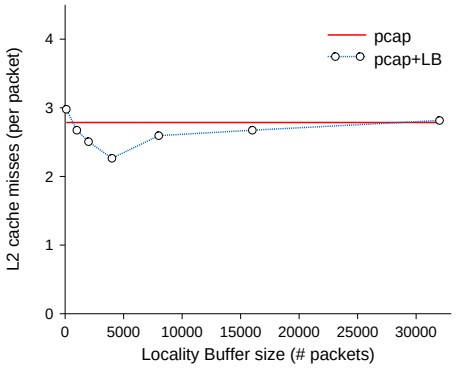


FIGURE 4.12: Fprobe's L2 cache misses as a function of the buffer size for 500 Mbit/s traffic.

mon is not so CPU-intensive as Snort, so it requires a larger amount of packets to be sorted in order to achieve a significant performance improvement.

We ran Appmon with traffic rates varying from 250 to 2,000 Mbit/s, observing similar results. Since Appmon does less processing than snort, less packets are dropped in high rates. The output of Appmon remains identical in all cases, which means that the periodic packet stream sorting does not affect the correct operation of Appmon's classification process.

4.3.4 Fprobe

`Fprobe` [3] is a passive monitoring application that collects traffic statistics for each active flow and exports the corresponding NetFlow records. We ran Fprobe with the original and our modified version of `libpcap` and performed the same measurements as with Appmon.

Figure 4.11 plots the CPU cycles and Figure 4.12 the L2 cache misses of the Fprobe variants for buffer sizes from 100 up to 32,000 packets, while replaying the trace at a rate of 500 Mbit/s.

We notice a speedup of about 11% in Fprobe when locality buffering is enabled, for 4,000 packets buffer size, while cache misses are reduced by 19%. The buffer size that optimizes the overall performance of Fprobe in this setup is around 4,000 packets. No packet loss occurred for Fprobe at all traffic rates. Fprobe is even less CPU-intensive than Appmon and Snort since it performs only a few operations per packet. The time spent in kernel for packet capturing is significantly larger than the time spent in user space. Thus, Fprobe benefits less from the locality buffering enhancements. For passive monitoring applications in general, the performance improvement due to locality buffering increases as the time spent in user space increases, and also depends on memory access patterns. Similar results were observed for all rates of the replayed traffic.

4.4 Discussion

We consider two main limitations of our locality buffering approach. The first limitation is that the packet reordering we impose results in non-monotonic increasing timestamps among different flows (it guarantees monotonic increasing timestamps only *per each* bi-directional flow). Therefore, applications that require this property, e.g., for connection timeout issues, may have problems when dealing with non-monotonic increasing timestamps. Such applications should either be modified to handle this issue, otherwise they cannot use our approach.

The second limitation of our approach is that our generic implementation within `libpcap`, which sorts packets based on source or destination port numbers, may not be suitable for applications that require a custom packet sorting or scheduling approach, e.g., based on application's semantics. Monitoring applications may perform similar processing for packets with specific port numbers, which cannot be known to the packet capturing library. Such applications should not use our modified `libpcap` version, but instead implement a custom scheduling scheme for packet processing order. Our implementation's goal, is to improve *transparently* the performance of a large class of existing applications, where the packet processing tasks depend mainly on the packets port numbers.

In particular, locality buffering technique is intended for applications which perform similar processing and similar memory accesses for the same class of packets, e.g., for packets of the same flow or packets belonging to the same higher level protocol or application. For instance, signature-based intrusion detection systems can benefit from locality buffering, due to different set of signatures matched against different classes of packets. Other types of monitoring applications may not gain the same performance improvement from locality buffering.

Finally, recent trends impose the use of multiple CPU cores per processor, instead of building and using faster processors. Thus, applications should utilize all the available CPU cores to take full advantage of modern hardware and improve their performance. Although L2 cache memory becomes larger in newest processors, more processor cores tend to access the shared L2 cache, so locality enhancements can still benefit the overall performance. Our approach can be extended to exploit memory locality enhancements for improving the performance of multithreaded applications running in multicore processors. Improving memory locality for each thread, which usually runs on a single core, is an important factor that can significantly improve packet processing performance. Each thread should process similar packets to improve its memory access locality, similarly to our approach, which is intended for single-threaded applications.

Applications usually choose to split packets to multiple threads (one thread per core) based on flow identifiers. A generic locality-aware approach for efficient packet splitting to multiple threads, in order to optimize cache usage in each CPU core, should sort packets based on their port numbers and then divide them to the multiple threads. This will lead to improved code and data locality in each CPU core, similarly to our locality buffering approach. However, in some applications

the best splitting of packets to multiple threads can be done only by the application itself, based on custom application's semantics, e.g., custom sets of ports with similar processing. In these cases, a generic library for improved memory locality cannot be used.

4.5 Summary

In this chapter, we presented a technique for improving the packet processing performance in a wide range of passive network monitoring applications by enhancing the locality of code and data accesses. Our approach is based on reordering the captured packets before delivering them to the monitoring application by grouping together packets with the same source or destination port number. This results to improved locality in application code and data accesses, and consequently to an overall increase in the packet processing throughput and to a significant decrease in the packet loss rate.

To maximize improvements in processing throughput, we combine locality buffering with memory mapping, an existing technique in `libpcap` that optimizes the performance of packet capturing. By mapping a buffer into shared memory, this technique reduces the time spent in context switching for delivering packets from kernel to user space.

We describe in detail the design and implementation of locality buffering within `libpcap`. Our experimental evaluation using three representative passive monitoring applications shows that all applications gain a significant performance improvement when using the locality buffering implementations, while the system can keep up with higher traffic speeds without dropping packets. Specifically, locality buffering results to a 25% increase in the processing throughput of the Snort IDS and allows it to process two times higher traffic rates without packet drops.

Using the original `libpcap` implementation, the Snort sensor starts to drop packets when the monitored traffic speed reaches 600 Mbit/s, while using locality buffering, packet loss is exhibited beyond 1 Gbit/s. Fprobe, a NetFlow export probe, and Appmon, an accurate traffic classification application, also exhibit significant throughput improvements, up to 12% and 18% respectively, although they do not perform as CPU-intensive processing as Snort.

Overall, we believe that implementing locality buffering within `libpcap` is an attractive performance optimization, since it offers significant performance improvements to a wide range of passive monitoring applications, while at the same time its operation is completely transparent, without the need to modify existing applications. Our implementation of locality buffering in the memory mapped version of `libpcap` offers even better performance, since it combines optimizations in both the packet capturing and packet processing phases.

5

Improving Accuracy Under Heavy Load

Over the past few years we have been witnessing an increasing number of security breaches and malicious activities in the Internet [155]. Network Intrusion Detection Systems (NIDSs) are crucial for the detection of security violations and suspicious activity, enhancing the robustness and secure operation of modern networks. However, the constant increase in link speeds and number of security threats poses significant challenges to NIDSs, which need to cope with higher traffic volumes and perform increasingly complex per-packet processing.

NIDSs operate in *soft real time*, meaning that under conditions of heavy traffic load the system will operate with degraded performance. When the network traffic load becomes higher than the peak processing throughput the NIDS can sustain, the CPU becomes saturated, and the Operating System inevitably starts dropping packets before delivering them to the NIDS, impeding its detection ability. Since these packets are not inspected, if they are part of an attack or other malicious activity, then that event will be missed.

Several techniques have been proposed for improving the performance of NIDSs by accelerating the packet processing throughput and thus processing higher traffic loads [13,35,84,157]. Other techniques automatically tune the NIDS configuration to balance detection accuracy and resource requirements [44,85]. However, given a highly loaded network, intrusion detection systems based on non-specialized hardware are usually not able to analyze all traffic to the desired degree [127]. Even after carefully tuning the NIDS according to the monitored environment, it will still have to cope with inevitable traffic bursts or processing spikes [135].

In this chapter, we present *selective packet discarding*, a technique that allows a NIDS to dynamically diagnose conditions of excessive traffic load and minimize their impact on its detection accuracy by choosing which packets should be dropped. Using selective packet discarding, the system selectively skips processing

packets that are less likely to affect the correct operation of the detection engine as soon as possible, instead of letting the Operating System randomly dropping arriving packets. This allows the NIDS to inspect a larger number of "useful" packets that are important to the detection process.

We observe that the first packets of a connection play a crucial role to the correct detection of a large class of attacks. For instance, signatures for threats like network service probes and reconnaissance attacks, brute force login attempts, protocol misbehaviour, and code-injection attacks, usually match packets that are among the first few hundred packets of a network flow. Moreover, the first control packets of a TCP connection are crucial to proper flow tracking and TCP stream reassembly, which are mandatory features of modern NIDSs [66,119]. If any of the packets in the TCP three-way handshake is lost, the corresponding flow will not be considered established, and potential attack vectors in this flow may evade detection. On the other hand, very large flows usually correspond to file transfers, P2P traffic, or streaming media applications, which typically are not related to security threats. Inspecting all packets from such "heavy-hitters," which comprise a large percentage of the total traffic, usually does not contribute much to the detection accuracy of a NIDS.

We implemented selective packet discarding in the Snort intrusion detection system [124] as a preprocessor that runs before the detection engine and all other preprocessors. It maintains state for the active flows and limits the number of packets that will be inspected per flow by discarding them from the rest preprocessors and Snort's core detection engine. It is also responsible to measure the delay and CPU usage when Snort is processing a group of packets and respectively adjust the number of packets that will forward for further inspection.

We experimentally evaluated our technique using production traffic, mixed with real attacks that Snort can detect. We replay the traffic at high speed rates using several traffic patterns, and we compare the detection accuracy of original Snort when packets are dropped with our modified Snort, with the selective packet discarding preprocessor enabled. Under overload conditions, the original Snort implementation misses a significant number of packets, resulting to a considerably lower number of alerts, with many of the labeled attacks in our trace passing undetected. This is a result of the random packet dropped by the Operating System. In contrast, selective packet discarding significantly improves the detection accuracy of Snort under increased traffic conditions, allowing it to detect most of the attacks that would have otherwise been missed.

The rest of this chapter is organized as follows: Section 5.1 introduces selective packet discarding and Section 5.2 provides the details of our implementation in Snort. In Section 5.3, we experimentally evaluate our technique under realistic conditions by replaying real traffic traces with different speeds. Finally, Section 5.4 summarizes this chapter.

5.1 Selective Packet Discarding

Ideally, an intrusion detection system should be able to capture and inspect all network traffic passing through the monitored link. In highly loaded networks, this may not be possible due to the limited computational power of the monitoring sensor. For traffic speeds higher than a few hundred Mbit/s, the system cannot process all monitored traffic, which unavoidably leads to packet drops [127].

One way to offload the detection engine is to select a subset of the monitored traffic to be excluded from the NIDS processing using a capture filter during initialization [91]. However, in a typical deployment, such a filter usually excludes only a small subset of the traffic, while events of excessive traffic load or bursty traffic can still occur. Given that under such conditions some packets will unavoidably get lost, we argue that it is better to proactively discard those packets that are less likely to affect the detection effectiveness of the system, instead of letting the OS drop packets at random.

In this section, we describe in detail the design and implementation of selective packet discarding, which dynamically controls which packets are going to get dropped in case of overload conditions with the minimal impact to the detection ability of the system. We first discuss *which* packets should be considered for discarding, and we propose a selection based on the flow size and the position of packets in their flows. Then, we describe the performance measurements that the NIDS should perform periodically to monitor the system's load and decide *when* selective packet discarding should be triggered. Finally, we present an algorithm that dynamically estimates *how many* packets should be dropped according to the system performance measurements.

5.1.1 Flow-based Packet Selection

The starting point of our work is the observation that in a typical NIDS, some network packets play a more important role for the detection of a large class of threats than the rest of the traffic, i.e., without processing these packets, there is an increased probability to miss an attack. For example, inspecting the protocol interactions of commonly targeted services like RPC and NETBIOS seems more important than inspecting a large file transfer of a file-sharing application.

Probably the most widely used abstraction when referring to network traffic, besides the network packets themselves, is the *network flow*. A network flow comprises packets with the same protocol, source and destination IP addresses, and source and destination port numbers (same 5-tuple) and represents a connection between two hosts.

The first packets of a network connection are very important for the correct detection of a large class of attacks. Many types of threats like port scanning, service probes and OS fingerprinting, code-injection attacks, and brute force login attempts, require a new connection for each attempt, and the attack vector is usually present in the first few thousands KB of the flow. By contrast, very large streams

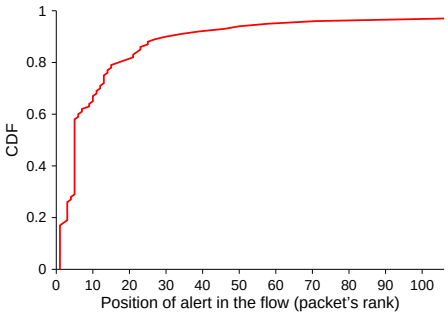


FIGURE 5.1: Distribution of the positions of matching packets within their flows.

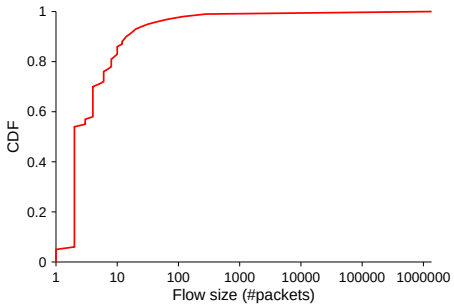


FIGURE 5.2: Cumulative distribution of flow size

usually correspond to file transfers, VoIP communication, or streaming media applications, which typically are not related to security threats. Very long network flows usually comprise a large portion of the total traffic in an organization's network, and inspecting the packets towards the end of such flows usually does not contribute much to the detection accuracy of a NIDS.

Another reason for the increased importance of the first packets of a connection is the flow tracking and TCP stream reassembly functionality of modern NIDSs. The packets in the three-way TCP handshake, which are always the first packets in a TCP flow, are crucial for updating the state of a new flow as it is established, identifying the direction of each stream, and performing TCP stream reassembly. If a control packet is lost during the connection initialization phase, the corresponding flow will not be considered as established and possible attack vectors present in subsequent packets of this flow may evade detection.

Analyzing the 9276 rules in the default rule set of Snort [7], we observe that 4627 of them contain the keyword `flow:established`, which defines that the detection engine should process the rest of the rule only if the packet belongs to an already established TCP connection. If under high load conditions a packet of the three-way handshake does not reach Snort's flow tracking preprocessor, then the rules that rely on flow tracking will never match for that flow, and potential attacks will not be detected.

Furthermore, attack vectors that span multiple packets in the beginning of the stream, such as the shellcode of a code injection attack or the URI of a malicious XSS HTTP request, are usually inspected after the original stream has been reassembled by the TCP stream reassembly preprocessor (Stream5 in Snort). If packets are being dropped randomly by the OS, the stream reassembly preprocessor may not receive a packet containing part of an attack, leaving the reassembled stream incomplete.

To verify our intuition based on the above observations, we analyzed traces of real attacks and extracted the actual position of the attack vector within the flow.

We ran Snort using real traffic captured at the access link of an educational institution network, which triggers 1976 alerts from 78 different rules. We further augmented the trace with 120 traces containing real attacks captured in the wild [117], which Snort detects using the default rule set. We interspersed these traces in random offsets within the large trace, so that the resulting trace generates a total of 2252 alerts due to 92 different attack signatures. Further details about the trace and the experimental environment are discussed in Section 5.3.1.

We slightly modified Snort to categorize packets into flows, and report the rank of the matching packet within its flow when it matches an attack signature. Figure 5.1 shows the cumulative distribution of the matching packet position within the flow for the 2252 alerts in the above trace. We observe that most of the alerts are triggered by the first few packets of a flow. For instance, 90% of the alerts were triggered within the first 30 packets of the flow, and only 3% of the alerts are triggered from packets coming after the 100th packet. This happens because there is a large class of malicious activities that most of the time takes place in the beginning of the flow. Such malicious activity includes port scans, attacks on the authorization and authentication mechanisms, usage of P2P applications (which often violates the corporate policy), as well as some protocol violations.

Flows usually follow a heavy tailed distribution on the Internet, i.e., the great majority of the flows have a quite small size, while only a very small subset has a very large size and is responsible for most of the total traffic volume [53]. Our trace also follows this property, as we can see in Figure 5.2, which shows the cumulative distribution of the flow sizes. We can see that 86% of the flows contain up to 10 packets, while 97% of the flows have no more than 70 packets. Only 0.4% of the flows have more than 1000 packets, which corresponds to 5972 out of the total 1,493,032 flows. There are also 74 flows (0.005%) with more than 100,000 packets. The average flow size in the trace is 50.2 packets.

Based on the above observations, we argue that a NIDS under high load conditions would benefit from focusing on the processing of the first packets of each flow, and discarding the rest. Selecting the packets to be processed by setting a flow size limit seems promising, since it will affect a small percentage of the flows, but will also exclude a large portion of the total traffic from processing. Dynamically setting the flow size cutoff limit according to the monitored traffic load is an important aspect of our approach, which we discuss in the rest of this section.

5.1.2 System Load Monitoring

Under conditions of excessive traffic load, in which the packet processing throughput of the NIDS is less than the monitored traffic throughput, the NIDS cannot process all the monitored traffic and unavoidably the OS kernel starts dropping packets. Implementing selective packet discarding requires the forecasting of overload conditions that will probably lead to dropped packets before the kernel actually starts dropping them. Captured packets are initially stored in a socket buffer in kernel space, before being copied to user space and then being delivered to the NIDS

through the packet capture library, in our case `libpcap` [92]. If the NIDS cannot consume the arriving packets fast enough, the socket buffer will fill up and excess packets will be dropped.

The main role of the socket buffer is to provide tolerance to short traffic bursts. In case of short bursts, packets will be buffered as long as the socket buffer has free space and will eventually be delivered for processing. The duration of the burst and the socket buffer size determine the number of packets that will be dropped, if any. In Linux, the `getsockopt` system call can be used to obtain the socket buffer size, which allows us to compute the minimum number of packets that can be buffered in case of a traffic burst by dividing the socket buffer size by 1500—the usual MTU for Ethernet. In all our experiments, we have set a large socket buffer size of 6MB, to rule out packet drops due to common small buffer configurations. If the amount of traffic is constantly higher than the NIDS processing throughput, packets will be dropped regardless of the socket buffer size, since it will be constantly full.

Our system identifies overload conditions using three metrics: (i) the occurrence of packet drops, (ii) CPU utilization approaching 100%, and (iii) a comparison between NIDS processing times and packet inter-arrival times.

Ideally, we would like to perform packet drops and CPU measurements at per-packet granularity. However, this would incur a prohibitively high overhead. Instead, we measure the NIDS's CPU usage, processing time, and packet drops once every N packets have been processed. We chose N based on the socket buffer size to permit the system to timely detect overloads conditions before any packet drops are caused by the kernel. We know that kernel can buffer at least this amount of packets in case of excessive load, before our discarding technique reacts. Additionally, N should be large enough to provide accurate measurements, according to the system's timing resolution [1]. Based on the above, in our setting with a 6MB socket buffer, we empirically set N to 5000.

Every 5000 packets, the system examines whether any packets were dropped by the kernel in the elapsed period using the `pcap_stats` function of `libpcap` [92]. Additionally, the system measures the user and system time using the `getrusage` system call, as well as the real time the NIDS spent while processing the previous group of N packets using `gettimeofday`. The CPU utilization of the elapsed period is computed as $(user\,time + system\,time)/real\,time$.

For the third metric, we compare the time t required for processing the group of N packets with the time interval s during which these packets were observed on the network. The processing time t corresponds to the $user\,time + system\,time$, while the time interval s is computed by subtracting the timestamp of the oldest packet in the group from the timestamp of the most recent packet. If $t > s$, then this is an indication that the kernel will probably start dropping packets, if not already. Otherwise, if $t < s$, the kernel did not drop any packets in that interval.

[1] In Linux, the `getrusage` system call provides 10ms resolution.

Since we need to predict packet loss events before the CPU gets saturated, we set an upper threshold for CPU usage and processing time, above which selective packet discarding is triggered. When $t > s$ and the CPU usage is relatively low, with no packet drop events during that period, the system decides whether more packets per flow should be processed or not based on a second lower threshold. These two thresholds should be close enough to allow for optimum resource usage and prevent CPU under-utilization. We also take into account the typical CPU load variation during short term intervals to avoid rapid oscillations, i.e, falling into a loop that would change very often the flow size limit up and down.

After running Snort using different traffic speeds and observing the correlation between the traffic load, CPU load, and packet drops, as we discuss in Section 5.3, we set the above thresholds as follows: the upper threshold is set to 0.95 and the lower threshold is set to 0.8, i.e., 95% and 80% CPU utilization, respectively. Putting it all together, the NIDS triggers selective packet discarding if during the processing of the previous N packets i) some packets were dropped by the kernel, or ii) $CPU\ utilization > 0.95$ and $t > 0.95s$. The condition for identifying an idle period is $CPU\ utilization < 0.8$ and $t < 0.8s$. Otherwise, the CPU utilization is within the desirable range and the flow size limits remain the same.

5.1.3 Flow Size Limit Adjustment Algorithm

Upon detection of an overload condition, the NIDS should back off and reduce the number of packets that it is going to process. First, we need to specify how many packets should be discarded, and then this number should be translated to the proper reduction of the per-flow cutoff limit. Ideally, the number of packets to be discarded should be such that it would allow the processing time for the packets in the following group to remain within the desirable range. In other words, the NIDS processing rate is N/t while the packets are coming with rate N/s We need to reduce the processing time t to become equal to $0.95s$, i.e., the number of packets to be discarded from the next group will correspond to processing time $t-(0.95s)$, that is:

$$(t - 0.95s)N/t \tag{5.1}$$

In case the system observes packet drops in that interval, we should also consider it in our decision. Therefore, the amount of packets that will be discarded is the maximum of i) the number of packets dropped in that interval, and ii) the number of packets estimated using Equation 5.1.

If an idle period is detected, the NIDS should ramp up and process more packets. The system computes the additional number of packets that should be processed in a similar manner as for packet drops. The NIDS can spend $(0.8s) - t$ more processing time in the next group of N packets, which corresponds to the following number of packets:

$$(0.8s - t)N/t \tag{5.2}$$

At this point, we have the mechanisms for estimating the number of packets that the NIDS should discard in case of overload. However, our selection strategy is based on limiting the flow size, which requires setting an appropriate flow size threshold. Based on the active flows in the monitored traffic, the system must estimate the reduction factor for the flow size threshold in order to skip processing the desirable number of packets. Similarly, in case more packets need to be processed, the flow size has to be increased appropriately by taking into account the sizes of the currently active flows.

The algorithm for deriving the new flow size limit for each interval, in case the above system measurements suggest so, is based on aggregated statistics the NIDS gathers during the classification of arriving packets into flows. The flow classification engine keeps packet counters for predefined flow size ranges, i.e., the system stores the number of packets that belong to flows with size from 0 to 100 packets, from 100 to 200 packets, and so on. The flow classification engine keeps these statistics using a table of 1000 integers. Each position i of the table indicates the number of packets that correspond to flows with size from $i * 100$ to $(i + 1) * 100$. The last position of the table counts the packets that belong to flows with size larger than 100,000 packets.

For each arriving packet, the classifier finds the flow in which the packet belongs to, increases the size of this flow by one (in terms of number of packets), and also increases the corresponding counter in the flow size statistics table. When a flow reaches a size x that is a multiple of 100, it levels up a range in the flow statistics table by subtracting from the $(x/100) - 1$ position the size of the flow in terms of number of packets, and adding it to the next position, $(x/100)$, of the flow statistics table. When a flow is closed, e.g., due to a proper TCP connection termination or due to some timeout of inactivity, the size of the flow is subtracted from the corresponding position of the flow statistics table.

Figure 5.3 presents the pseudocode of the flow size limit adjustment algorithm. In case of packet discarding, the algorithm descends the flow statistics table (lines 8–11), starting from the range of the current flow limit (lines 6–7), and counts the packets that will be discarded in each lower flow size range, until we reach the desirable number. The procedure for increasing the flow size limit is similar, by ascending the flow statistics table (lines 18–21) until the required number of packets is encountered. Then, the flow size limit is adapted accordingly (lines 12–13, 22–23).

5.2 Implementation within Snort

5.2.1 Selective Packet Discarding

We have implemented the selective packet discarding approach within the Snort [124] intrusion detection system as a preprocessor configured to run before the detection engine and all other preprocessors. The preprocessor receives each packet immediately after Snort's Layer-4 packet decoding, and based on the protocol, source and

```
update_CPU_usage();
update_dropped_packets();

if (dropped_packets || (CPU_usage>0.95 && t>0.95*s)) {
  reduce_packets = max(dropped_packets, (t-(0.95*s))*N/t);
  if (limit == NO_LIMIT) range = 1000;
  else range = limit/100;
  while (reduce_packets>0 && range>0) {
    reduce_packets -= flowstats[range];
    range--;
  }
  if (range == 0) limit = 10;
  else limit = range*100;
}
else if (cpu_usage<0.8 && t<0.8*s) {
  increase_packets = ((0.8*s)-t)*N/t;
  range = limit/100;
  while (increase_packets>0 && range<1000) {
    range++;
    increase_packets -= flowstats[range];
  }
  if (range == 1000) limit = NO_LIMIT;
  else limit = range*100;
}
```

FIGURE 5.3: Simplified pseudocode for the flow size limit adjustment algorithm.

destination IP addresses, and, in case of TCP/UDP packets, source and destination port numbers, it looks up the corresponding flow through a hash table. For each flow, the preprocessor keeps statistics about its size in number of packets. Based on the flow size and the current packet discarding flow size limit, the preprocessor decides whether the current packet should be discarded, or forwarded to the other Snort preprocessors and the core detection engine.

Furthermore, as we have discussed in Section 5.1.3, the preprocessor keeps the aggregate number of captured packets for predefined flow size ranges. Flows are closed either after a timeout of inactivity (set to 10 seconds in our experiments) or due to normal TCP protocol connection termination after RST of FIN/ACK packets. It is important to precisely follow TCP connection terminations and discard the relevant flow statistics, in order to prevent attackers to evade detection by closing and opening new TCP connections immediately. Thus, each new connection will be considered as new flow and its first packets will always be processed by Snort.

5.2.2 System Performance Monitoring

The second important function of the preprocessor is the flow size limit readjustment according to the algorithm presented in Section 5.1.3. As we have described,

the algorithm is activated every N packets, since system performance measurements must be performed in N packet intervals. The size of the interval, N, is automatically chosen based on the size of the socket buffer and the systems' resolution in measuring CPU time. After processing N packets, the preprocessor reasons about potential overload conditions based on the performance measurements and adjusts the flow size limit accordingly.

5.3 Experimental Evaluation

In this section, we present the experimental evaluation of our prototype implementation of selective packet discarding in the Snort IDS. We first examine the effect of the flow size limitation in Snort's performance, using Snort for offline trace inspection. Then, we experimentally evaluate our technique by replaying the mixed trace at several high rates, and we compare the detection accuracy of original Snort when packets are dropped due to overload with our modified Snort with the selective packet discarding preprocessor enabled to gracefully adapt to the same overload conditions.

5.3.1 Experimental Environment and Traffic Used

Our experimental environment consists of two PCs interconnected through a 10 Gbit switch. The first PC is used for traffic generation, which is achieved by replaying real network traffic traces at different rates using `tcpreplay` [145]. The traffic generation PC is equipped with an Intel Xeon 2.00 GHz CPU with 6 MB L2 cache, 2 GB RAM, and a 10 Gbit network interface. This setup allowed us to replay traffic traces with speeds up to 900 Mbit/s. Achieving larger speeds was not possible using large network traces because usually the trace could not be effectively cached in main memory.

By rewriting the source and destination MAC addresses, the generated traffic is sent to the second PC, the intrusion detection sensor, which captures the traffic and inspects it using the original Snort, as well as our extended version with selective packet discarding. We modified Snort v2.8.3.2, used the latest official rule set [7] containing 9276 rules, and enabled all the default preprocessors as specified in its default configuration. The NIDS PC is equipped with an Intel Xeon 2.66 GHz CPU with 4 MB L2 cache, 2 GB RAM, and a 10 Gbit network interface. The kernel socket buffer size was set to 6 MB in order to minimize packet drops due to short packet bursts. Both PCs run 64bit Ubuntu Linux (kernel version 2.6.27).

For the evaluation we used an anonymized one-hour long trace captured at the access link that connects an educational network with thousands of hosts to the Internet. The trace contains 58,714,906 packets, corresponding to 1,493,032 different flows, totalling more than 40 GB in size. As already discussed in Section 5.1.1, for this trace Snort generates 1976 alerts from 78 different rules using the default rule set.

Reasoning about whether these alerts are true positives or not would require manual inspection of each alert and the corresponding matching packets. Most of the matching rules are related to common threats such as probes for vulnerable web applications and database servers, old buffer overflow exploits, and protocol violations. Such traffic is typically received by DMZ servers. There are also quite a few alerts from rules that look for suspicious activity, such as `robots.txt` access or `HTTP 403 Forbidden` responses, which correspond to 49 and 263 alerts, respectively. Given the nature of the triggered alerts, we believe that most of them are true positives. However, based on our experience, and since we have not checked all alerts one by one, we speculate that some of the alerts must be false positives. In order to strengthen our evaluation, as discussed in Section 5.1.1, we augmented the trace with 120 short traces of real attacks, adding 276 more alerts from 14 different rules which are definitively true positives.

5.3.2 Flow Cutoff Impact Analysis

In our first experiment, we explore the impact of imposing a limit in the number of packets of each flow that are going to be processed on Snort's processing throughput and detection accuracy. We modified our preprocessor to discard the packets of each flow after a certain flow size limit has been reached. We ran Snort using different flow cutoff values using the augmented network traces described in the previous section. Snort loads the trace for offline analysis and our preprocessor allows only packets which lay before the maximum allowed flow size limit to be inspected by Snort's detection engine. Since this an offline analysis, there is no dynamic adaptation in the flow cutoff size—the same flow size limit is used for the whole duration of each run.

For each run, we measure Snort's execution time using the `time` Unix tool, which provides us the elapsed user, system, and real time. The processing throughput is measured as the total trace size divided by the user plus system time. We repeat each measurement 10 times and report the average value of the throughput for different flow size limits. For each run, the detection accuracy is defined as the percentage of alerts triggered for each different flow cutoff value, divided by the total number of alerts (2252) which we know a priori that are triggered by the trace. We are mostly interested in the detection of the 276 attacks that we have injected in the trace, since we know that the corresponding alerts are definitively true positives, but it is also desirable to observe as many of the rest of the alerts in the trace as possible.

Figure 5.4 presents Snort's throughput and detection accuracy when varying the number of processed packets per flow from 10 to 100,000. The unmodified version of Snort achieves a throughput of 560 Mbit/s. When enabling the preprocessor, as the number of inspected packets per flow decreases the throughput is increased, since Snort inspects fewer packets. For instance, for a flow cutoff limit of 1000 packets, Snort can process up to 1400 Mbit/s of traffic, while with 100 packets per flow the processing throughput reaches 2 Gbit/s. When using larger

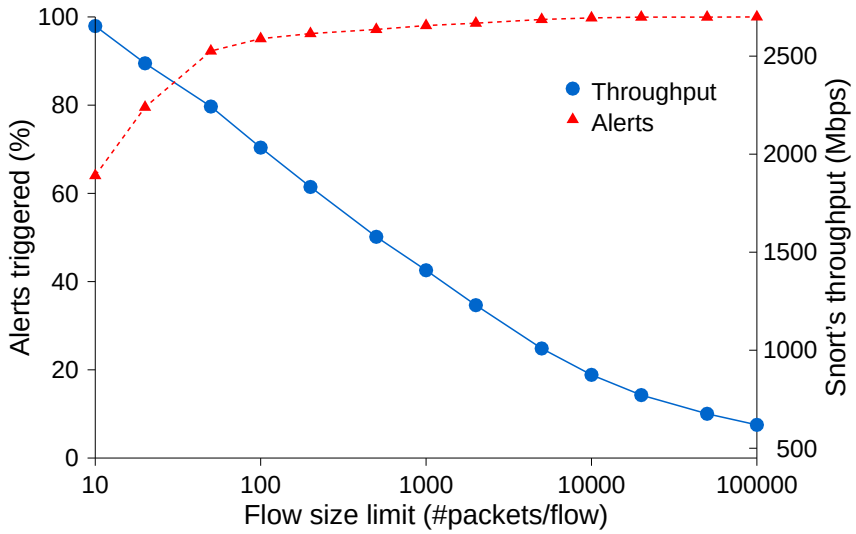


FIGURE 5.4: Snort's throughput and detection accuracy as a function of the flow size limit.

flow cutoff sizes, the throughput approaches the unmodified Snort's throughput, e.g., with a limit of 50,000 packets the throughput drops to 675 Mbit/s, which still is a 20% improvement.

As the flow cutoff size increases, the number of triggered alerts also increases, i.e., the number of missed events decreases. For flow limits higher than a few hundreds of packets, only a small percentage of alerts is missed. As already expected from Figure 5.1, alerts are triggered mostly due to packets that belong to the first few packets of a flow. For instance, processing up to 10,000 packets per flow results to 5 missed alerts out of the 2252 alerts in the trace. while at the same time the throughput increases 56%. For a cutoff size of 50,000 packets only one alert was missed. The 276 alerts due to the real attacks that we manually injected are all triggered even for a cutoff limit as low as 20 packets per flow.

Even when inspecting just the first 100 packets of each flow, 95% of the alerts are still triggered. Considering the corresponding improvement in Snort's throughput, which is 3.62 times faster reaching up to 2033 Mbit/s, enabling selective packet discarding for traffic volumes higher that 560 Mbit/s seems promising. As we are going to see in the next section, under such conditions, the packet drops by kernel result a much higher number of missed alerts. When the monitored traffic throughput drops to normal and Snort is not high loaded, the selective packet discarding preprocessor will dynamically adapt the flow size limit as much as effectively disabling packet discarding at all.

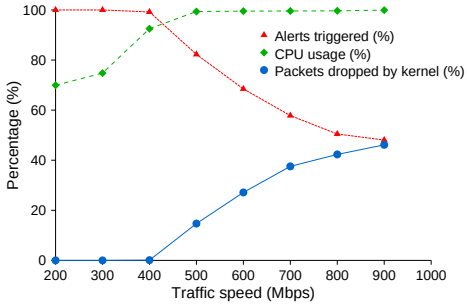


FIGURE 5.5: Performance of unmodified Snort as a function of the monitored traffic speed.

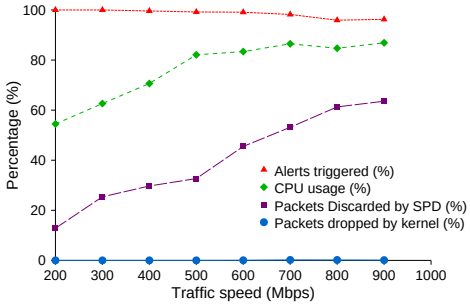


FIGURE 5.6: Performance of Snort with selective packet discarding as a function of the monitored traffic speed.

5.3.3 Improving Detection Accuracy under High Load

We now evaluate the detection accuracy of unmodified Snort and our extended Snort version with selective packet discarding under realistic conditions of increased load.

Figure 5.5 shows the performance of original Snort when replaying traffic with speeds varying from 200 to 900 Mbit/s. For each traffic speed, we repeated the measurements 10 times and report the average of the percentage of triggered alerts, the CPU utilization of Snort for processing the packets, and the percentage of dropped packets by the OS. We see that for speeds higher than 500 Mbit/s, a significant percentage of packets is dropped by the kernel, ranging from 15% for 500 Mbit/s up to 46% for 900 Mbit/s traffic. When packets are dropped, the CPU utilization is always higher than 99%, since Snort cannot handle the high traffic volume.

The consequence of these drops is a significant reduction in the number of detected events. For a traffic speed of 500 Mbit/s, with just 15% of the packets being randomly dropped by the OS, Snort misses 18% of the alerts, i.e., only 1852 of the total 2252 alerts are reported. When 46% of the packets are dropped, for 900 Mbit/s traffic, about half of the alerts are missed. Even for 400 Mbit/s traffic, a slight percentage of dropped packets (0.096%) causes 16 alerts to be missed (0.7%). Furthermore, among different runs for the same traffic speed, Snort generates different sets of alerts, indicative of the non deterministic results that random packet drops induce.

Moreover, the 276 alerts due to the real attacks we injected are lost with the same probability as all other alerts in the trace. For instance, for 500 Mbit/s traffic, Snort identified 223 out of the 276 attacks, missing 19% of the alerts. For 900 Mbit/s, just 55% of these alerts were successfully detected. These results demonstrate that Snort's detection accuracy degrades significantly under conditions of excessive traffic load.

Figure 5.6 shows the performance of Snort with selective packet discarding enabled. A first observation is that with selective packet discarding, the number of packets dropped by the kernel is negligible. There are no packet drops for traffic speeds up to 600 Mbit/s, while there is just a 0.098% of dropped packets for 900 Mbit/s. We also notice that for high traffic speeds, the CPU utilization remains within the desirable range imposed by the 0.8 lower and 0.9 upper thresholds. Figure 5.6 also shows the percentage of packets that are selectively discarded according to the size of the flow in which they belong to. As we expected, the percentage of discarded packets increases according to the traffic speed. For instance, in 500 Mbps traffic speed 32% of the packets are selectively discarded from inspection, in order to prevent Snort overloading. In 900 Mbps, the percentage of discarded packets reaches 63%. By discarding the desirable amount of packets according to the traffic load, Snort controls the CPU utilization and keeps it constantly within the desirable range.

The number of selectively discarded packets is larger than the number of dropped packets by the kernel in unmodified Snort for the same speeds. There are two explanations for this outcome. First, the selective packet discarding algorithm is purposely quite aggressive in discarding packets in order to proactively predict and prevent packet drops from the kernel. Thus, the preprocessor tends to discard more packets from the end of the flows and benefit from preventing uncontrolled random packet drops from the kernel. Second, in unmodified Snort, packets are dropped in kernel level, before they are copied to user level. With selective packet discarding, all packets are first delivered in user space and then are discarded by the Snort preprocessor, which results to a higher number of discarded packets. However, even with eventually less inspected packets, selective packet discarding allows Snort to achieve a much better detection accuracy, as discussed below. Moreover, the number of flows affected by selective packet discarding is just 0.42% of the total number of flows for the highest traffic speed of 900 Mbit/s, and even smaller for lower traffic speeds. In contrast, random packet drops by the kernel affect a significantly higher number of flows.

Finally, Figure 5.6 shows the significant improvement in detection accuracy by enabling selective packet discarding. For all traffic rates, even for 900 Mbit/s, Snort reports almost all of the alerts that exist in the monitored traffic. For 500 Mbit/s traffic, our modified Snort reports 2234 out of the 2252 alerts (99.2%), which is an improvement of 20% over unmodified Snort. The percentage of triggered alerts remains almost constant as the traffic speed increases, falling slightly to 96.3% for 900 Mbit/s traffic, missing just 84 events.

Since we cannot be sure about the nature of all triggered alerts in the trace, we study separately the detection accuracy of the 276 attacks we manually injected in the trace. Figure 5.7 presents the number of alerts triggered by our modified Snort for each traffic speed divided in two categories: the 276 alerts due to the injected attacks, and the rest 1976 alerts due to the real traffic of the original trace. We observe that for all traffic speeds, Snort was able to detect all the real attacks that we manually inserted in the traffic, suggesting that selective packet discarding

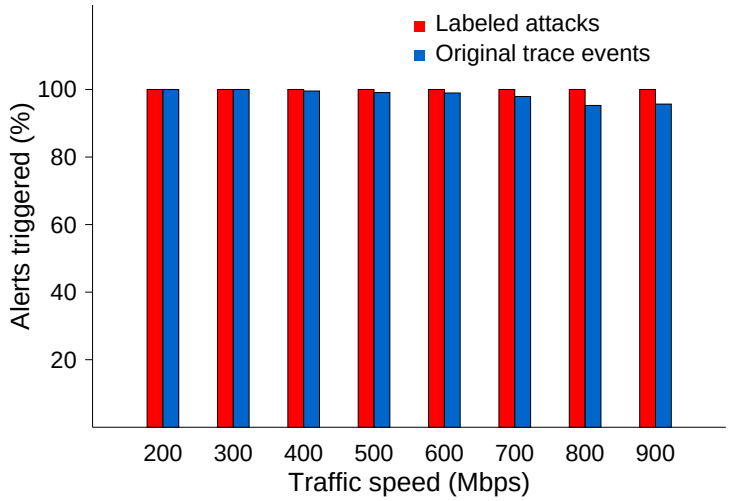


FIGURE 5.7: Alerts triggered by modified Snort as a function of the traffic speed.

indeed tends to improve the detection accuracy of real attacks. Of course, the vast majority of the alerts due to events in the original trace are still triggered.

5.4 Summary

Events of excessive network traffic load are a common fact that affects the performance of intrusion detection systems. Under conditions of heavy traffic load or sudden traffic bursts, the processing throughput of the system cannot cope with the amount of traffic that needs to be inspected, and the OS unavoidably drops excess arriving packets at random.

In this chapter, we presented selective packet discarding, a best effort approach that gracefully reduces the amount of traffic that reach the detection engine of the NIDS by selectively discarding packets that are less likely to affect its detection accuracy. We have implemented selective packet discarding in the Snort NIDS as a preprocessor that constantly measures performance aspects of the system in order to detect overload conditions and dynamically adjusts the number of packets that needs to be discarded. This is achieved by setting a cutoff limit to the number of packets to be inspected for each network flow.

A concern that arises when using selective packet discarding is that a sophisticated attacker could exploit the flow size limit and evade detection by filling the stream with benign requests and then send the actual attack vector after the flow cutoff limit has been reached. Although such an attack may be feasible for protocols like HTTP, which allows multiple requests to be sent through the same connection, other services terminate the connection after the end of each transaction, especially in case of protocol violations or failed requests. Furthermore, for proto-

cols that support persistent connections, such repetitive behaviour can be detectable by following the protocol's request/response semantics. However, without selective packet discarding, an attacker can evade detection from an overloaded NIDS by repeating the attack multiple times—depending on the traffic load, after a certain number of attempts the attack will go undetected. Selective packet discarding makes such overload attacks harder to achieve.

Our experimental evaluation with real-world traffic and labeled attacks demonstrates that selective packet discarding improves significantly the detection accuracy of Snort under increased traffic load conditions, allowing it to detect most of the attacks that would have otherwise been undetected.

6
Tolerating Overload Attacks

Network traffic monitoring systems are increasingly used to improve the performance and security of modern computer networks. These monitoring systems have always been depended on an efficient and reliable underlying packet capture mechanism. However, such network traffic monitoring systems are now called to operate in an unpredictable and sometimes hostile environment where transient traffic and malicious attackers may easily overload them up to the point where they cease to function correctly. Unfortunately, traditional packet capturing systems, have not been designed for such hostile environments and do not gracefully handle overhead conditions. For example, when faced with overload conditions and full packet queues, most packet capturing systems start to discard all incoming packets for as long as the overload persists and until it resolves itself. We believe that this naive approach to packet discarding, which surprisingly is still being used by most network traffic monitoring systems, has three major disadvantages:

- It may drop packets which contain *important information*, such as an attack or a particular pattern.

- It can be exploited by attackers to *hide their attack*: the attackers can flood the system with bogus packets up to the point where the system overloads and starts discarding (i.e., not inspect) most of the incoming packets [27, 33, 34, 119, 135]. Then, attackers can send their attacks being almost sure that they will evade the monitoring system.

- It robs monitoring applications from the opportunity to *selectively discard the unimportant packets* in the traffic [86, 87, 89, 107], and forward for processing and further inspection the *important* ones.

To cope with high traffic volumes, several techniques have been proposed for improving the performance of network intrusion detection systems (NIDSs) by accelerating the packet processing throughput [13, 35, 84, 157]. However, it is not clear how these approaches cope with existing algorithmic overload attacks, and more importantly with *future* and potentially *unknown* algorithmic attacks that may exist in monitoring applications but have just not been exposed in the public domain yet. Other techniques automatically tune the NIDS configuration to balance detection accuracy and resource requirements [44, 85]. However, given a highly loaded network, intrusion detection systems based on non-specialized hardware are usually not able to analyze all traffic to the desired degree [127]. Even after carefully tuning a NIDS according to the monitored environment, it will still have to cope with inevitable traffic bursts or unpredictable algorithmic complexity attacks.

Existing solutions for algorithmic complexity attacks are based on algorithmic improvements, which consider worst case performance comparing with the average case [135]. However, software is still written so that it is vulnerable to such attacks. Thus, monitoring applications and NIDSs have to defend against complexity attacks that have not been seen in the wild yet. In general, it is not clear how to find and defend against all sources of algorithmic complexity attacks.

To address these problems we propose *Selective Packet Paging (SPP)*: a new approach for mitigating both traffic overloads and algorithmic attacks by exploiting the following two dimensions:

- We introduce a *new level in the memory hierarchy* of packet capturing systems: a level which is able to store all packets during periods of traffic or algorithmic overload.

- We propose a *randomized timeout algorithm* which is able to detect malicious network packets that trigger algorithmic overload attacks and isolate them for further disk processing.

We observe that the root of packet loss in modern packet capture systems is the limited number of packets that the operating system kernel can store in a memory buffer. When this buffer fills up, the next incoming packets will be just discarded until the overload resolves itself. To address this issue, we propose a new memory management system called Packet Paging. Contrary to the traditional single-layer memory management systems, Packet Paging exposes a two-layer memory hierarchy: (i) the first layer is stored in the main memory of the computer and contains the most recently received packets. The size of this layer is usually large enough to handle all incoming traffic at line speed when there is no transient or algorithmic overload. (ii) The second layer is stored in the local disk storage system and is used mostly to accommodate packets which can not fit in the first layer. During an overload, when the first layer fills up, incoming packets continue to be stored at the second layer until the overload condition resolves itself, or is resolved by clever choices of selective packet discarding or human intervention. Modern disk

systems have enough capacity to store several hours of traffic. We believe that this time is more than enough to allow human intervention to resolve the problem.

In case of carefully crafted packets which exploit an algorithmic complexity vulnerability, Packet Paging will buffer all the excessive packets to disk, both benign and crafted. Moreover, the crafted packets will be processed in-order and will eventually slowdown the system. Thus, the completion time for processing benign packets will be affected significantly, even if no packet is lost. To address this issue, we have extended Packet Paging with the SPP technique. Based on randomized timeouts, SPP is able to detect crafted packets that result in slow processing rates, and to remove these packets from the critical path. Instead of dropping these packets, we buffer *them* to secondary storage and we give them lower priority, so they will be processed only when the system has the necessary resources. In this way, the processing time of the rest packets is not affected.

We implemented Packet Paging and SPP techniques within the popular libpcap [92] packet capturing library, so that a large class of existing deep packet inspection systems can benefit from our approach transparently without any code modifications. The Packet Paging implementation is based on two basic ideas: (i) we give priority to a *packet storing* thread over a *packet processing* thread and (ii) while a memory buffer is full, we store packets to disk. The (i) ensures that all packets will be stored with no losses, while with (ii) we avoid the overheads of disk I/O in normal cases without overloading. Moreover, to maintain the correct order in which packets were arrived, we design an indexing structure which points to the location of the next packet that should given, in memory or disk. We also aim to optimize disk throughput by writing and reading packets in batches. The implementation of SPP is based on randomized timeout values and tracking the number of packets processed in each timeout.

The main contributions of SPP are:

- We demonstrate that the root of packet discarding under overload in the existing packet capture systems is the poor design choices in memory management.

- We propose Selective Packet Paging, a two-layer memory management system that is able to store practically all network packets during overloads: long enough to allow human intervention to solve the problem. SPP is also able to efficiently resolve algorithmic complexity attacks by detecting and removing from the critical path any malicious packets that slowdown a monitoring system.

- We implement our system and integrate it with the libpcap packet capture library [92].

- We experimentally evaluate our approach using the Snort NIDS [124], and we show that it is able to sustain algorithmic attacks and traffic overloads

without discarding any packets. Contrary, the traditional packet capture approach is forced to discard the largest percentage of the incoming packets and force Snort to miss 100% of the attacks.

- We analytically evaluate the randomized timeout selection approach of SPP and show that the probability of detecting an algorithmic complexity attack reaches certainly exponentially fast.

The rest of this chapter is organized as follows: Section 6.1 introduces the design of SPP and Section 6.2 provides details of our prototype implementation within libpcap. In Section 6.3 we present analytical and simulation-based evaluation for the detection capabilities of SPP using a randomized timeout. In Section 6.4, we experimentally evaluate our techniques under algorithmic complexity and traffic overload attacks. We show that the Snort NIDS is vulnerable to these attacks, while SPP achieves significant tolerance against processing and traffic bursts. Finally, Section 6.5 discusses alternative choices, and Section 6.6 summarizes this chapter.

6.1 Selective Packet Paging

The main cause of packet loss during overloads, is usually the limited number of packets that the Operating System's packet capturing subsystem can store in main memory. Thus, in case of traffic overloads or algorithmic attacks the main memory fills up pretty quickly and the rest of the incoming packets are just dropped. One obvious solution to this problem would be to increase the main memory available to the packet capturing subsystem. Unfortunately, typical main memories can not store more than a few minutes of network traffic for a high-speed link. Thus, an algorithmic attack or a network overload that lasts for more than a few minutes will eventually lead to packet drops and to reduced system functionality.

In modern systems, the available disk storage, typical few TBs, is up to three orders of magnitude larger than the available storage capacity in main memory, which is typically few GBs large. Thus, captured packets can be buffered on disk for several hours under overload conditions, instead of just a few seconds in main memory. For instance, a system with 4 GB of RAM and 4 TB of disk storage monitoring an 1 Gbit/s line, can buffer about 32 seconds of traffic in main memory and up to about 9 hours of traffic in disk storage.

6.1.1 Multi-level Memory Management

In this work we propose to break away from the single-level memory hierarchy traditionally used by packet capturing subsystems and employ a multi-level memory hierarchy consisting of at least two levels: a main memory and a secondary storage. Under normal circumstances captured packets are written in main memory. Under traffic overload or algorithmic attacks, when the main memory fills up, extra packets are written to secondary storage.

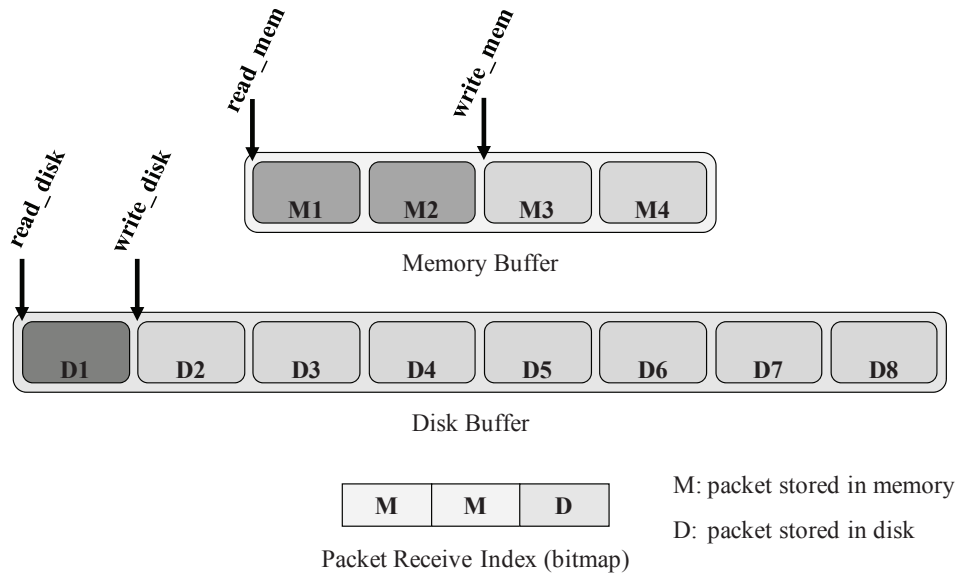


FIGURE 6.1: A snapshot of Packet Paging for buffering packets to memory and disk. The Packet Receive Index indicates that the first two packets are stored in the Memory Buffer while the third packet is in the Disk Buffer.

Figure 6.1 presents the two-level memory hierarchy of our approach. The first layer (i.e., the memory buffer) is organized as a circular queue. As long as the buffer is not full, newly arriving packets are written in the memory buffer. When the main memory buffer fills up, newly arriving packets are stored in the second layer of the memory hierarchy, i.e., the disk buffer. This buffer is also organized as a circular queue. Note that while newly arriving packets are being written to disk, main memory space is being freed up since monitoring applications will consume existing packets. In this case, we would like to be able to write newly arriving packets in main memory and thus avoid the disk access overheads. Indeed, our system *first* tries to write incoming packets to main memory and only if this is full, it tries to write them to disk. However, this choice implies that sequentially arriving packets may be written to different levels of the memory hierarchy, oscillating between main memory and disk.

For example, suppose three packets (i.e., p1, p2, and p3) arrive in the system and that main memory can hold only one of them. Then, packet p1 will be written to main memory and packet p2 will be written to disk. Now, assume that while p2 is written to the disk, the monitoring application consumes one more packet from main memory and therefore creates space in main memory; thus p3 will now be written in main memory. At the end, packet p1 will reside in main memory, packet p2 will reside in disk and packet p3 will reside in main memory. This example indicates that we need somehow to record the order of arriving packets.

For this reason we keep a Packet Receive Index which keeps strictly in FIFO order the location of all incoming packets. To deliver packets in the correct order, we use one bit for each incoming packet in the Packet Receive Index, as shown in Figure 6.1. This bit indicates whether the next packet to be delivered was stored in main memory or on disk.

6.1.2 Randomized Timeout Intervals

Although multi-level memory management makes sure that no packets are lost during an overload, algorithmic attacks may force the CPU to spend most, if not all, of its time on processing bogus attack packets that just trigger an algorithmic overload—benign network packets will just keep accumulating on the disk. Selective Packet Paging advocates that instead of blindly sending subsequent packets to secondary storage when the main memory is full, we should develop mechanisms to detect packets that trigger an algorithmic overload, weed them out, and send *them* to secondary storage for processing at a later point in time. In this aspect, they will free-up the CPU which can then be dedicated to processing of the rest of the network packets.

To detect those packets that trigger an algorithmic overload attack one could make use of the CPU timestamp counter: read the timestamp counter before and after processing of each packet. If the packet's processing time in higher than a threshold, then this packet can be considered as an algorithmic attach packet, due to its unusual high processing time. However, this approach has a significant drawback: the attack packet cannot be evicted from the inspection engine, as it is detected after its processing ends, so it will result in a system slowdown. Also, with the timestamp counter the total system's time is measured, not only the time spent at the DPI application. Thus, this approach is susceptible to false positives.

An alternative approach to detect those packets is to use a timeout counter (i.e., a timer): when the monitoring application starts processing a new packet, a timeout counter is initialised to a timeout value.[1] If the timeout counter expires while the monitoring application is still processing the same packet, then the packet is considered suspicious. Otherwise, if the packet processing completes before the timeout counter expires, the application starts processing the next in line packet and the counter is reset to its timeout value. This way, attack packets can be evicted from the inspection engine and will not delay the system. Also, the time spent only on the process of the DPI application is measured. Unfortunately, however, setting and resetting timeout counters at each and every packet could impose a large processing overhead, especially when timeouts are implemented inside the kernel, and especially when the network traffic is dominated by small packets.

To reduce this overhead, SPP argues that we should not set the timeout counter at each and every packet. Instead, the counter should be set at *periodic* intervals: if the packet being processed during the interval expiration is the same packet that

[1]The value of the timeout counter should be larger than the processing time of ordinary packets and smaller than the processing time of algorithmic complexity attack packets.

was being processed at the time the counter was set, then SPP considers this packet as suspicious. As a result, the packet, along with all subsequent packets from the same network flow, will be buffered to disk. Although setting the timeout counter at periodic intervals has the potential to reduce the timeout processing overhead, especially if the intervals are long enough, choosing an appropriate timeout value can be really tricky: a very large timeout value may miss a lot of attack packets, while a very small value may impose a large processing overhead on the system. To make matters worse, a single predefined timeout value (or a deterministic sequence of timeout values) could theoretically be evaded by a sophisticated attacker who manages to send all attack packets between successive timeouts.

To solve this problem, SPP uses a randomized timeout interval. That is, instead of choosing a predefined constant timeout, SPP chooses a timeout which is a random variable uniformly distributed in the interval $[low, high]$. Obviously, the average value of a timeout is $(high + low)/2$, which influences the overhead associated with timeout processing. Choosing a large value for $high$ reduces the (average) timeout overhead, while choosing a small value for low makes detection of algorithm attacks easier. Indeed, to make sure that they avoid detection, attackers should only send attack packets that trigger algorithmic complexity attacks that last for no more than low seconds. Therefore, a small low value forces a (what used to be) sophisticated algorithmic complexity attack to degenerate into a brute force Denial of Service attack consisting of a torrent of attack packets, which can be easily detected and filtered out.

6.2 Implementation

We implemented Selective Packet Paging within the popular packet capturing library libpcap [92], so that a large class of existing network monitoring applications can benefit from SPP without any code modifications. libpcap in Linux uses the PF_PACKET socket, which receives all packets from a network interface card. Each packet is first stored in memory allocated by the kernel for DMA transfer, and then copied to user-level accessible memory. In our prototype implementation we use three separate threads: (i) the *packet capturing and storing thread*, which receives packets from kernel and stores them to memory or on disk if there is no space in memory; (ii) the *packet processing* thread, which finds the next packet through the Packet Receive Index, and calls the callback function registered by the monitoring application through `pcap_loop()` for processing each packet, or returns the packet's data and header through `pcap_next()`; and (iii) the *disk I/O thread*, which handles all communication with the secondary storage. We give higher priority to the packet capturing and storing thread over the packet processing thread, to ensure that all packets will be stored during overloads. To optimize disk throughput, the disk I/O thread transfers packets between main memory and disk in batches. Moreover, to avoid delays from blocking read operations, the disk I/O thread prefetches the next batch of packets from disk to memory.

Packet capturing and storing thread:

```
if (suspicious_flow(next_pkt))
   write(next_pkt, disk_buffer, low_priority);
else if (memory_buffer.size==FULL)
   write(next_pkt, disk_buffer, normal_priority);
else
   add(next_pkt, memory_buffer);
```

Packet processing thread:

```
if (packet_index.next==NULL) {
   disable_timer();
   current_pkt=read(disk_cache, low_prioriy);
   process_pkt(current_pkt);
   enable_timer();
}
else if (packet_index.next==MEMORY) {
   current_pkt=read(memory_buffer);
   process_pkt(current_pkt);
}
else if (packet_index.next==DISK) {
   current_pkt=read(disk_cache, normal_priority);
   process_pkt(current_pkt);
}
pkt_counter++;
```

Timer expiration handler:

```
if (pkt_counter==prev_pkt_counter) {
   buffer(current_pkt, low_priority);
   mark_suspicious_flow(current_pkt);
}
prev_pkt_counter=pkt_counter;
set_timer(low+rand()%(high-low));
```

FIGURE 6.2: Pseudocode for the implementation of Selective Packet Paging with a randomized timeout interval.

Figure 6.2 presents the pseudocode for implementing SPP with a randomized timeout. The packet capturing and storing thread receives all packets from the NIC and stores them in memory or on disk. If a packet belongs to a flow that has been marked as suspicious, it is stored immediately on disk in a file with low priority packets. Else, if the memory buffer is full, the packet is stored on disk in a separate file with normal priority packets.

The packet processing thread selects the next packet for processing. In case there are packets with normal priority, the next packet is selected from memory or disk based on the packet indexing structure. If there is no such packet, then a packet with low priority is selected for processing. In the latter case, the timer is disabled or the timeout interval is increased, otherwise the timer could expire again while this packet is being processed.

The processing thread keeps a counter of the processed packets. When the timer expires, it checks how many packets have been processed from the previous timer expiration. If the number of processed packets remains the same, then the current packet delays the system for an unreasonably long time. Thus, the packet is evicted and buffered to disk, while its flow and source IP address are marked as suspicious. Packets belonging to suspicious flows are also written to disk as low priority packets.

The next timer interval is scheduled to a random time between the low and $high$ limits.[2] The low value is related to the normal processing times: it should be set slightly higher than the worst-case processing time of a benign packet[3]. The $high$ limit controls the overhead for setting and expiring a timer, and the detection probability. Larger values result in lower overhead and lower detection probability per packet, i.e., more time to detect a stealthy algorithmic overload attack.

The timer expires based on the time passed while only the current process (or the system on behalf of the current process) is executing, so SPP is not affected by external background activities. Thus, the time passed between two successive timer expirations was spent only within the packet processing thread. To avoid false positives, a proper value for the low limit should be used. Then, only packets with significant processing delays will be detected as suspicious. But even in case of false positives, packets will not be dropped. They will follow a different data path, and they will be eventually processed when the system has the available resources.

6.3 Analytical Evaluation

Using a random timeout uniformly distributed in the range $[low, high]$, Selective Packet Paging makes it difficult for attackers to evade detection, while keeping the timeout overhead reasonably low. Indeed, a very large value for $high$ keeps the average timeout value (i.e., $(low + high)/2$) reasonably large, and thus the overhead reasonably low, while a small value for low forces the attacker to lean towards sending packets that trigger short algorithmic attacks: shorter in duration than low. Since, however, the timeout is a random variable, it is theoretically possible even for an attack packet that triggers a long algorithmic attack to evade detection. This is especially true if the timeout interval chosen during the time the attack packet is being processed is relatively large. In the rest of this section we show that although it is theoretically possible for one attack packet to evade detection, it is very unlikely that several attack packets will go undetected. An attacker who wants to sustain an algorithmic attack has to send several attack packets, and it is improbable that none of them will be detected. In the rest of this section we will estimate

[2]The low and $high$ limits can be set either directly by the user, or automatically by the monitoring application, e.g., by profiling the normal processing times when the system is not under attack.

[3]In this context, we note as benign packets the normal packets that do not aim to attack the network monitoring application, and as attack packets the crafted packets that try to impede its correct operation.

the probability that SPP detects an attack with a single timeout choice, and then the average number of timeout intervals that SPP will need to detect the attack.

To simplify our analysis, we initially assume that there are only attack packets, that each attack packet is being analyzed for a constant interval of d microseconds, and that $low < d$. Selective Packet Paging is able to detect an attack if two successive timeouts expire within the same interval d for the same attack packet. The first timeout expires at time t_1, which will fall within an interval i of an attack packet. Thus, $i \times d < t_1 < (i+1) \times d$. The probability that the second timeout, which expires at time t_2, will also fall within the interval i is:

$$P(t_2 < (i+1) \times d) = \frac{d - t_1 - low}{high - low} \tag{6.1}$$

since there are $high - low$ possible choices for a timeout but only $d - t_1 - low$ accepted choices so that the second timeout expires within the interval i. In the unfortunate for the attacker case that t_1 falls in the beginning of the interval i, there are $d - low$ accepted choices for the second timeout. In case that t_1 falls in the position $(i+1) \times d - low - 1$ of the interval i, there is only one accepted choice for the second timeout: the low timeout value. On average, there are $(d - low)/2$ accepted choices for the second timeout in case that the first timeout t_1 falls within the first $(d - low)$ values of the interval i. If t_1 falls in the last low values of the interval i, there is no accepted choice for the second timeout. Thus, on average over the whole interval i, there are $(d - low)/2 \times (d - low)/d + 0 \times low/d$ accepted choices for the second timeout, for each accepted choice of the first timeout.

Overall, the probability for detection with two timeouts in the same interval is:

$$\begin{aligned} P(det) &= \frac{(high - low) \times (d - low)/2 \times (d - low)/d}{(high - low) \times (high - low)} \\ &= \frac{(d - low)^2}{2 \times d \times (high - low)} \end{aligned} \tag{6.2}$$

since the possible choices for two timeouts are $(high - low) \times (high - low)$, the accepted choices for the first timeout are $(high - low)$, and the accepted choices for the second timeout are $(d - low)/2 \times (d - low)/d$. The probability of not detecting an attack after N timeouts have expired is $(1 - P(det))^N$, and thus the probability of detecting the attack after N timeouts is $1 - (1 - P(det))^N$: we see that the detection probability approaches 1 very fast as N gets larger. Also, the detection probability from Equation 6.2 means that on average SPP will need $T = 1/P(det) + 1 = (2 \times d \times (high - low)/(d - low)^2) + 1$ timeouts to detect the attack. This number corresponds on average to $T \times (high - low)/(2 \times d)$ attack packets and $T \times (high - low)/2$ microseconds.

The outcomes of our analysis are also valid in case that attack packets induce variable delays with an average delay of d microseconds. In a more realistic scenario there would be both benign and attack packets, so that the attack packets

would be a percentage a of the total packets, with $0 < a < 1$. The average processing time for a benign packet is t microseconds, and we expect that $t < d$. In this case the detection probability from Equation 6.2 is:

$$P(det) = \frac{a \times d}{d + t} \times \frac{(d - low)^2}{2 \times d \times (high - low)} \tag{6.3}$$

since the probability of the first timeout to expire within an interval of an attack packet is $a \times d/(d+t)$. As the percentage a of the attack packets and the difference $d - t$ of the processing times between attack and benign packets increase, the probability of Equation 6.3 approaches the probability of equation 6.2.

6.3.1 Comparison with Simulation Results

To validate our analysis for the detection capabilities of Selective Packet Paging with a randomized timeout, we perform a simulation-based evaluation and compare the results with our analytical evaluation. Figure 6.3 presents the detection time in milliseconds as a function of the processing time of each attack packet, based on both our analysis and our simulation study, for two attack scenarios: i) when all packets are attack packets, and ii) when the percentage of attack packets is 25%.

The processing time t of each benign packet is uniformly distributed between 1 and 30 microseconds, with an average value of 15 microseconds, while the processing time d of each attack packet is constant for each simulation. In our simulations we vary d from 100 to 1000 microseconds, to examine how the detection time will be affected. The randomized timeout for SPP takes values from low=50 to $high$=1000 microseconds.

We simulate the processing times of benign and attack packets, according to the above parameters, and we continuously set a timer randomly between the specified low and $high$ timeout limits. When two successive timeouts expire during the processing interval of the same attack packet, the experiment ends and outputs the time passed for the detection. We repeated each experiment for one million times and we report the average values.

For the analytical evaluation we used the probability from Equation 6.3 to compute the number of timeouts T needed for the detection:

$$T = 1/P(det) + 1 = \frac{2 \times d \times (high - low) \times (d + t)}{(d - low)^2 \times a \times d} + 1 \tag{6.4}$$

Thus, the average detection time is $T \times (high - low)/2$ microseconds.

In Figure 6.3 we can see that simulation results are very close to the expected results based on our analysis. We observe that SPP with the randomized timeout is able to detect even attacks with very small delays within just a few milliseconds. For instance, when the processing time of an attack packet is 200 microseconds, SPP detects the attack within the first 10 ms in case that all packets belong to this attack. In a more conservative attack, where only 25% of the total packets impose 100 microseconds processing time, SPP needs about 170 ms to detect and resolve

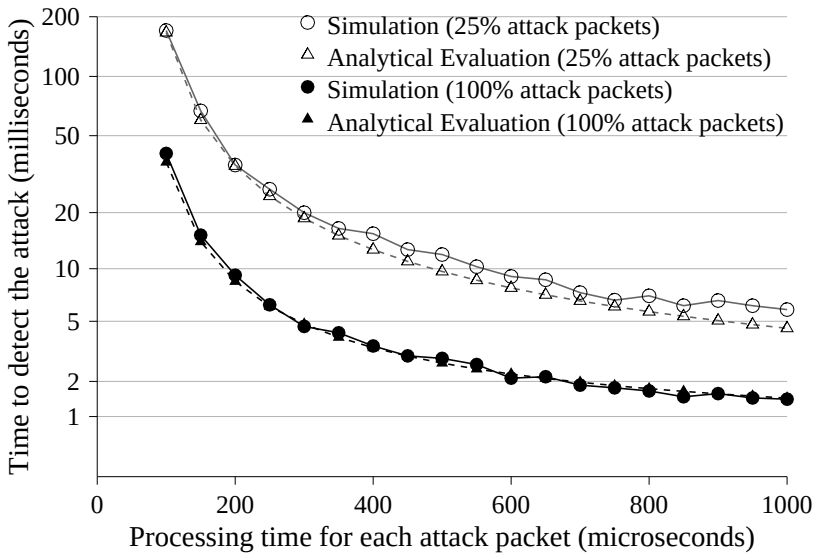


FIGURE 6.3: Detection time as a function of the processing time of the attack packets. We observe that SPP is able to detect the attack within a few milliseconds in most cases.

the attack. However, such a conservative attack for a period of a few milliseconds will not affect significantly the system. More aggressive attacks, which can harm significantly the monitoring system, are detected by SPP within less than 2 ms.

6.4 Experimental Evaluation

In this section we present experimental results when running the Snort NIDS with SPP under overload. We first describe the experimental environment (Section 6.4.1), and then we evaluate the performance of SPP during an algorithmic complexity attack (Section 6.4.2) and a traffic overload attack (Section 6.4.3), comparing with the original libpcap.

6.4.1 Experimental Environment

The Hardware

Our experimental environment consists of two PCs interconnected through a 10GbE switch. The first PC is used for traffic generation, which is achieved by replaying real network traffic and traces at different rates using `tcpreplay` [145]. The traffic generation PC is equipped with two dual-core Intel Xeon 2.66 GHz CPU with 4 MB L2 cache, 4 GB RAM, and a 10GbE network interface. By rewriting the source and destination MAC addresses in all packets, the generated traffic is

sent to the second PC, the intrusion detection sensor, which captures the traffic and processes it using the Snort NIDS, with the original libpcap for packet capturing as well as our modified version of libpcap with SPP. The NIDS PC is equipped with two quad-core Intel Xeon 2.00 GHz CPUs with 6 MB L2 cache, 4 GB RAM, and a 10GbE network interface. Beyond the system disk, we equipped the NIDS PC with four 750 GB 7200 RPM SATA disks, organized in RAID 0 using the Linux software RAID `mdadm` utility, resulting to 3 TB total storage. Both PCs run 64bit Ubuntu Linux (kernel version 2.6.32).

The parameters

The size of the memory mapped buffer between kernel and user level for storing packets, when the original libpcap is used, is set to 1 GB. Thus, the original system is able to tolerate very short processing or traffic spikes, using the 1 GB memory buffer. Note that, for fairness, the total main memory used in our Selective Packet Paging system is equal to 1 GB as well: 500 MB socket buffer size and a 500 MB memory buffer within the modified libpcap. In addition to a main memory of 1 GB, SPP also uses a secondary storage of 3 TB allocated (and distributed) on the four dedicated magnetic disks. This amount of storage, with the increased disk throughput achieved using the RAID 0 scheme, aims to provide significantly better tolerance for prolonged algorithmic attacks and traffic overloads.

We use the `ext3` file system, with the default ordered journaling mode. Using `ext2` or `ext3` with any other journaling mode results in very similar performance, since we operate only on a single large file. We run the Snort NIDS [124] version 2.8.3.2, using the latest official Sourcefire VRT rule set [7] containing 9276 rules, and enable all the default preprocessors as specified in its default configuration. Note that, since the original Snort implementation is single-threaded, it can not benefit from the underlying multi-core processor. To have a fair comparison with SPP, we scheduled all three threads of the SPP implementation to run on a single CPU core. Obviously, letting all three threads run on different cores would significantly improve performance even further.

The traces

For the evaluation we use four sets of traces, summarized in Table 6.1. As background traffic, we use an anonymized one-hour long trace (named `T1`) captured at the access link that connects a large university campus with thousands of hosts to the Internet. The trace contains 58,714,906 packets, corresponding to 1,493,032 different flows, totalling more than 46 GB in size. The average traffic rate in the trace is about 110 Mbit/s. We replay this trace at the actual rate that it was captured, as real background traffic. We do not consider the alerts produced by Snort from this trace's traffic in our analysis.

Trace	Packets	Replay rate
`T1`: background traffic	58.7 M	110 Mbit/s
`T2`: crafted packets	1,000	2 Kbit/s–2 Gbit/s
`T3`: real attacks	7,616	1 Mbit/s
`T4`: traffic bursts	1.5 M	1 Mbit/s–2.5 Gbit/s

TABLE 6.1: Traces used in our experiments.

The second trace (named `T2`) is used in the first experiment in Section 6.4.2 to trigger an algorithmic overload in Snort. `T2` contains synthetically created packets which exploit the backtracking vulnerability of a regular expression used in a Snort rule, resulting to significant slowdown. The third trace (named `T3`) consists of 120 short traces containing real attacks captured in the wild [117]. Snort detects these attacks using the default rule set, resulting to 276 alerts from 14 different rules. We replay this trace continuously in parallel with the second trace, and measure the percentage of these 276 alerts that Snort was able to detect when using the original libpcap and our SPP system. In the second experiment in Section 6.4.3, instead of `T2` we replay a short part of `T1` (named `T4`) at higher rates, in parallel with `T3`, to measure the percentage of alerts that Snort detects under traffic overload conditions.

Disk Throughput

In our first set of experiments we set out to measure whether SPP is able to sustain the storage of packets at line speed during a prolonged overload situation. As expected, the bottleneck is caused by the disk storage system. Thus, we first measure the throughput at which our RAID 0 system can read and write data in our experimental setup. Table 6.2 shows the read and write performance of our disk system as measured by the `bonnie` benchmark [22]. We see that the RAID 0 system is able to sustain 3 Gbit/s of read throughput and 1.8 Gbit/s of write throughput.

	Single Disk	RAID 0
Seq. read throughput (Mbit/s)	808	3072
Seq. write throughput (Mbit/s)	466	1790

TABLE 6.2: Throughput of single-disk and multi-disk storage system in our experimental environment, measured by the `bonnie` benchmark.

6.4.2 Algorithmic Complexity Attack

Attack Description

In this experiment we perform an algorithmic complexity attack against the Snort NIDS, similar to the attack described by Smith et al. [135]. Specifically, our attack targets the performance of regular expression matching based on the PCRE library [67]. PCRE represents regular expressions using a tree-like structure. For a given input string, PCRE iteratively explores paths in this structure until it finds an accepting state, in which case it declares a match. If it fails to find a match, it backtracks and tries another path until all paths have been explored. As the number of backtracks increase, more time is spent for PCRE matching, and the overall performance is decreased. Thus, we send a number of crafted packets to Snort targeted to a specific rule with a vulnerable PCRE regular expression, so that the packets' payload result in a large number of backtracks.

To prevent performance problems and denial of service through PCRE overload, Snort can be configured with a limit on the number of backtracks it performs per each regular expression. However, this limit can lead to missed alerts and evasion attacks as well: by cutting short the PCRE backtracks, a rule may not be triggered on payload that would cause a Snort alert. Attackers may create packets with payloads resulting to more backtracks than the specified limit before the complete match, thus evading detection. In our experiments we had disabled the PCRE backtracking limits.

In this experiment, the attack targets the SMTP Snort rule 2682, which detects an attempt to exploit a known vulnerability of Internet Explorer that results in e-mail attachment execution, by sending an incorrect MIME header. [4] This rule is matched against TCP packets destined to port 25, belonging to an established TCP connection, based on a PCRE expression, as shown below:

```
alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25
(msg:"SMTP spoofed MIME-Type auto-execution attempt";
flow:to_server,established; content:"Content-Type|3A|";
nocase; content:"audio/"; nocase;
pcre:"/Content-Type\x3A\s+audio\/(x-wav|mpeg|x-midi).*
filename=[\x22\x27]?.{1,221}\.(vbs|exe|scr|pif|bat)/smi";
metadata:service smtp; reference:bugtraq,2524;
reference:cve,2001-0154; reference:url,www.microsoft.com/
technet/security/bulletin/MS01-020.mspx;
classtype:attempted-admin; sid:3682; rev:5;)
```

[4] A more detailed analysis of algorithmic complexity attacks based on several vulnerable Snort rules is presented in [135]. Except PCRE, the Aho-Corasick string matching algorithm [9] was also found to be vulnerable to excessive backtracking attacks in Snort, when it uses nondeterministic finite automata (NFAs). A detailed analysis of the number of rules which are vulnerable to algorithmic complexity attacks is out of the score of this work. However, we speculate that there will almost always be rules written in a way that algorithmic complexity attacks could be possible.

The PCRE expression in this rule searches initially for the *"Content-Type"* string anywhere in the packet, followed by the byte 0x3A and one or more white characters, and then for the *"audio/"* string followed by *"x-wav"*, *"mpeg"* or *"x-midi"* strings. Then, any sequence of characters is acceptable before the *"filename="* string is found, optionally followed by the byte 0x22 or 0x27. PCRE will try to match the largest possible sequence of characters. Next, any sequence from 1 up to 221 characters is again acceptable, looking for the largest possible match. Finally, if one of the five stings *".vbs"*, *".exe"*, *".scr"*, *".pif"* or *".bat"* is also found after all the previous matches, an alert will be triggered.

This regular expression gives us the ability to produce a large number of backtracks in a 1500-byte packet. As any character sequence is acceptable between the *"audio/"* and *"filename="* strings, and up to 221 characters before the last string match, multiple instances of each string needed for a match, except the last one, can be inserted into the packet's payload in order to increase the possible paths that will be traversed while searching for a match. This way, the number of backtracks will increase exponentially high. We call such PCRE rules as *vulnerable* rules to algorithmic complexity attacks. For instance, consider the following payload:

```
Content-Type0x3A audio/mpegContent-Type0x3A audio/mpeg...
Content-Type0x3A audio/mpegfilename=0x22filename=0x22
filename=0x22filename=0x22...filename=0x22filename=0x22.ba
.ba.ba...ba.ba.ba.ba...filename=0x22filename=0x22.ba.ba...
.ba.ba.ba.ba.ba...filename=0x22filename=0x22.ba.ba.ba...ba.
.ba.ba.ba.ba.ba.ba.ba
```

The PCRE engine will first try to find a match from offset 0, where the first *"Content-Type"* string is found. Then, it will pass all the input till the last *"filename="* instance. It will continue until the end of the input, with no match found. Then, PCRE will backtrack once for each previous *"filename="* instance in the payload to search for a match starting from offset 0. Since no match will be made, PCRE will then repeat the same process starting from each *"Content-Type"* instance. Thus, the number of backtracks will be approximately equal to the number that *"filename="* string appears multiplied by the number of *"Content-Type"* strings.

Based on this pattern, we created 1500-byte packets belonging to an established connection destined to port 25 (trace `T2`). When processed by Snort, each crafted packet results in approximately $21,120$ backtracks during PCRE maching, and in a processing time about 1360 times slower than the average processing time for benign SMTP packets in trace `T1`. We use PCRE and this vulnerable rule as a proof-of-concept experiment to demonstrate the impact of an algorithmic complexity attack in a network monitoring application, and the performance benefits offered by SPP to tolerate such attacks.

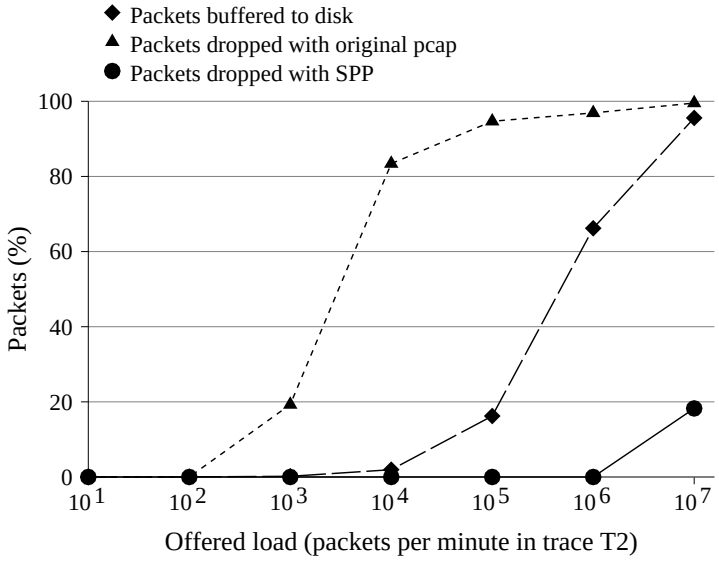


FIGURE 6.4: Percentage of dropped packets and packets buffered to disk as a function of the offered load (packets per minute). We see that as soon as the offered load exceeds merely 10^2 packets per minute, the original libpcap starts losing packets, and when the load becomes 10^4 packets per minutes, it drops more than 80% of the packets. On the contrary, SPP sustains zero loss all the way up to 10^6 packets per minute. Indeed, only when the offered load increases beyond that, and reaches the limit of the disk's write throughput, only then, SPP starts to lose about 18% of the incoming packets.

Results

Packet loss. In this experiment we set out to explore what is the packet loss of the original libpcap system and our SPP system during an algorithmic overload attack. The setup of the experiment is as follows:

1. We replay trace `T1` as background traffic at its original rate (110 Mbit/s).
2. We replay trace `T2` at a variable rate: we start with a rate as low as 10 packets per minute and increase it all the way up to 10^6 packets per minute. Recall that packets in this trace are specially crafted to trigger an algorithmic attack.
3. We transmit trace `T3` continuously at a rate of 1 Mbit/s for the entire duration of the experiment. We run each experiment for 10 minutes.

Figure 6.4 presents the percentage of the packets dropped by Snort when running on top of the original libpcap and when running on top of SPP. We observe that when the offered load (i.e., trace `T2`) reaches a mere 10^3 packets per minute, Snort on top of the original libpcap starts losing packets, and when the offered

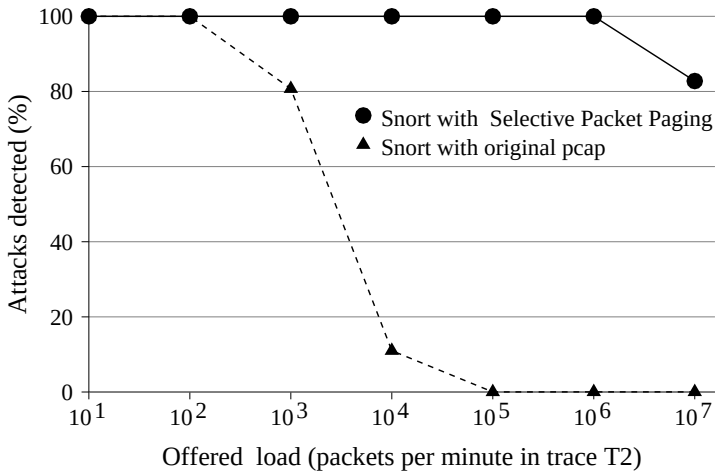


FIGURE 6.5: Percentage of detected attacks as a function of the offered load (the higher the better). We see that as soon as the offered load exceeds just 10^2 packet per minute, the original libpcap system starts to lose packets and attacks. As the offered load increases, the performance deteriorates. When the offered load exceeds 10^5 packets per minute, the original libpcap system is able to detect zero attacks. On the contrary, SPP manages to sustain 100% detection rate for loads as high as 10^6 packets per minute.

load exceeds 10^4 it loses more than 80% of the packets. On the contrary, at these loads, SPP loses no packets and manages to store them to disk. Figure 6.4 also presents the percentage of packets buffered to disk with SPP. We see that the packets buffered to disk by SPP are fewer than the packets dropped by libpcap at similar rates. This is because by identifying and weeding out algorithmic attack packets, SPP is able to dedicate more CPU cycles to processing ordinary packets.

These packet losses are directly translated to undetected attacks, i.e., attacks which have evaded the Intrusion Detection System. Indeed, in Figure 6.5 we show the percentage of attacks detected by the two systems. We see that Snort on top of the original libpcap starts missing attacks as soon as the offered load exceeds 10^2 packets per second, and loses all attacks as soon as the load reaches 10^5 packets per second. Thus, to evade detection with probability 99.98%, an attacker has to send 10^5 crafted packets per minute, exploiting the slowdown from a large number of backtracks when matching the specific regular expression. Fortunately, at these load rates, SPP does not miss any of the attacks as all packets are stored to secondary memory and are eventually inspected by the intrusion detection system.

When using SPP, all attack attempts are detected for up to 10^6 packets per minute. That is, an attacker needs to send about 10^7 packets per minute to reduce the probability of being detected just by 17%. In this extreme case, our disk system was not able to store all incoming packets due to the high traffic rate. Compared

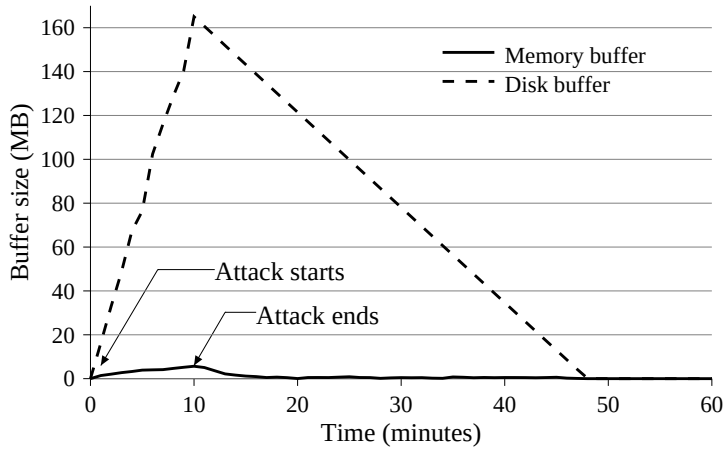


FIGURE 6.6: Size of memory and disk buffers over a 60-minute time period when sending 10,000 crafted packets/minute for the first 10 minutes.

to the original libpcap, SPP can handle 10,000 times more crafted packets. Thus, SPP offers significant tolerance to highly efficient algorithmic complexity attacks.

Recovery time. To measure the time that the system needs to recover from overload, we performed the following experiment:

- We sent trace `T1` at its normal rate for 60 minutes.
- We sent trace `T2` at a rate of 10^4 packets per minute for the first 10 minutes of the experiment.
- We sent trace `T3` at an 1 Mbit/s rate for 60 minutes.

Figure 6.6 presents the size of memory buffer and disk buffer over time. We report the size of each buffer every one minute. We observe that with SPP the size of the memory buffer was always less than 6 MB, for the whole 60-minute period. The attack packets (and their associated flows) identified by SPP were sent to disk. Indeed, to accommodate the attack packets, the disk buffer size increased from 16.23 MB (at minute 1) to 165 MB (at minute 10, which was the highest point of the attack), and then slowly decreased back to zero at minute 48. We should emphasize that throughout the experiments no packet was lost. Thus, significantly less packets are stored to disk in case of SPP, since only the attack packets are buffered. The attack packets remain to disk and they are processed with slow rates till minute 48, when the system completely recovers from the attack.

6.4.3 Traffic Overload

Attack Description

In this set of experiments we set out to explore how SPP responds to traffic bursts. To do so, we run Snort as in the previous section and feed it with the following traffic:

1. Trace `T1` is sent at its original rate and serves as background traffic.

2. At each minute we send a traffic burst that lasts for 30 seconds, using the traffic from trace `T4`. The peak rate of the burst is varied throughout the experiments from 1 Mbit/s up to 2.5 Gbit/s to evaluate how the intensity of the bursts may influence SPP.

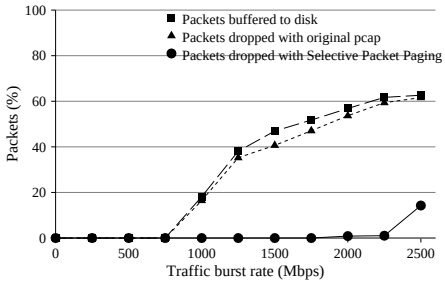
3. Trace `T3` that contains 276 real attacks is sent continuously at 1 Mbit/s for the entire duration of the experiment. Each experiment lasts 10 minutes.

Results

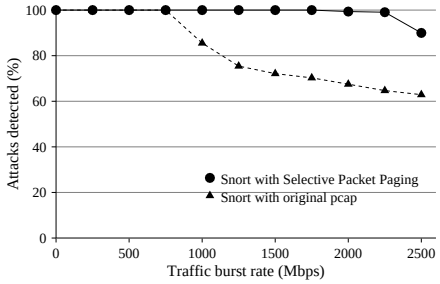
Traffic bursts of constant duration. Figures 6.7(a), 6.7(b), and 6.7(c) present the percentage of dropped packets, the percentage of detected attacks, and the average delay per packet, as a function of the rate of traffic bursts. We observe that Snort on top of the original libpcap starts dropping packets when the traffic bursts are around 1 Gbit/s, resulting in about 17% undetected attacks. When the bursts reach a rate as high as 2 Gbit/s, 53% of the packets are dropped and 32.5% of the attacks are missed. On the other hand, Snort with SPP drops no packets and misses no attacks even at rates as high as 2 Gbit/s. Although our disk system writes packets with 1.5 Gbit/s throughput, the two-level memory hierarchy allows to store up to 2 Gbit/s without packet loss. Only when the burst rates exceed 2.25 Gbit/s the secondary storage is not able to keep up with network traffic and SPP starts to lose packets.

It has been argued that in some cases, it is better to drop some packets so as to deliver the rest of them without delay. In this aspect, one might prefer to use the original libpcap (which drops packets rather than delaying them) rather than using a version that may delay packets longer. Figure 6.7(b) shows that dropping packets may force the system to miss a significant percentage of attacks: as many as 40%.

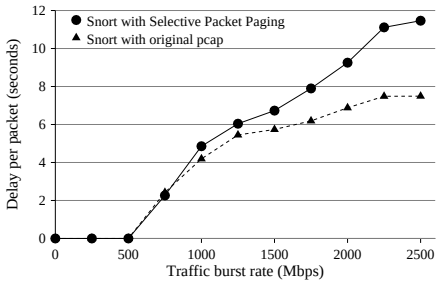
In Figure 6.7(c) we show that even the approaches that choose to drop some packets to avoid delaying all of them, do not necessarily reduce delays significantly. Indeed, we see that with a traffic rate as high as 2 Gbit/s, SPP delivers all packets to Snort for inspection within 9.25 seconds on average: just 25% slower than the original libpcap system, which delivers less than half of the packets. Overall, we believe that dropping about half of the packets in order to deliver the other half about 25% faster, much like libpcap does, seems not like a trade-off that monitoring applications would choose to make.



(a) Dropped packets and packets buffered to disk as a function of the traffic burst rate for 30-second bursts.



(b) Detected attacks as a function of the traffic burst rate for 30-second bursts.

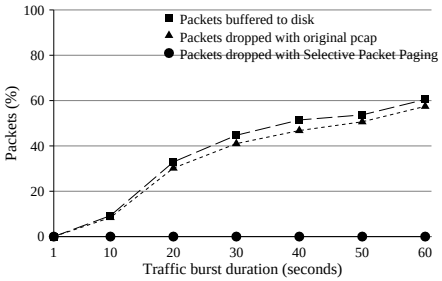


(c) Average delay for delivering packets to Snort as a function of the traffic burst rate for 30-second bursts.

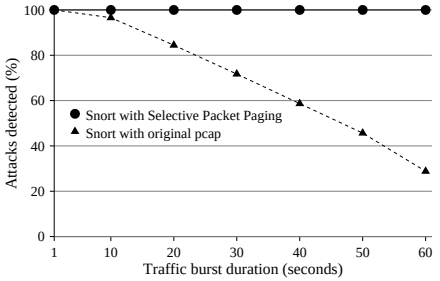
FIGURE 6.7: Performance of SPP and original libpcap in case of 30-second bursts as we vary the traffic burst rate.

Traffic bursts of variable duration. Figures 6.8(a), 6.8(b), and 6.8(c) show the detected attacks, packet drops, and average delay per packet while varying the duration of traffic bursts from 1 to 60 seconds with a constant 1.5 Gbit/s traffic burst rate. Each traffic burst is repeated every minute for 10 minutes. We see that as the duration of traffic burst increases, the original libpcap drops more packets and Snort misses more attacks. For traffic bursts lasting 40 seconds at a time, 47% of the packets were dropped due to CPU utilization and only 58% of our injected attacks were successfully detected, while for traffic bursts lasting 60 seconds, 57% of the packets were dropped and just 29% of the attacks detected.

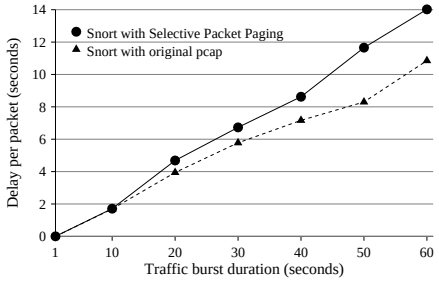
Fortunately, SPP tolerates the 1.5 Gbit/s traffic burst even at 60 seconds duration, i.e., when 1.5 Gbit/s traffic is sent continuously for 10 minutes. All packets are buffered to disk successfully and are inspected with a slight increase in delay. In case of 60 seconds duration, the average delay per packet with SPP is 14 seconds, while the original libpcap delivers only the 43% of packets with a delay of 10.8 seconds. The rest 57% of the packets are dropped: they are never delivered.



(a) Percentage of dropped packets and packets buffered to disk as a function of the traffic burst duration for 1.5 Gbit/s bursts.



(b) Percentage of detected attacks as a function of the traffic burst duration for 1.5 Gbit/s bursts.



(c) Average delay for delivering packets to Snort as a function of the traffic burst duration for 1.5 Gbit/s bursts.

FIGURE 6.8: Performance of SPP and orginal libpcap in case of 1.5 Gbit/s traffic rate as we vary the burst duration.

Recovery time. In Figure 6.9 we present the size of memory and disk buffers when sending 30-second traffic bursts with 1.5 Gbit/s rate for 10 minutes, and continue sending only background traffic for another 50 minutes. We report the size of each buffer once per minute. The memory buffer remains full at 500 MB for the first 12 minutes, while the disk buffer size increases continuously during the first 10 minutes all the way up to 21.3 GB. From minute 11 to minute 13, the disk buffer size is reduced from 21.3 to 3.5 GB, since the system's resources are sufficient to process the excessive packets buffered during the traffic bursts. Thus, in the 14th minute, both memory and disk buffers are empty, so the system has fully recovered from the traffic overload attack. Compared with the algorithmic complexity attack, the system recovers faster from traffic overload, in this experiment within four minutes, because packets are not maliciously crafted to further slowdown Snort.

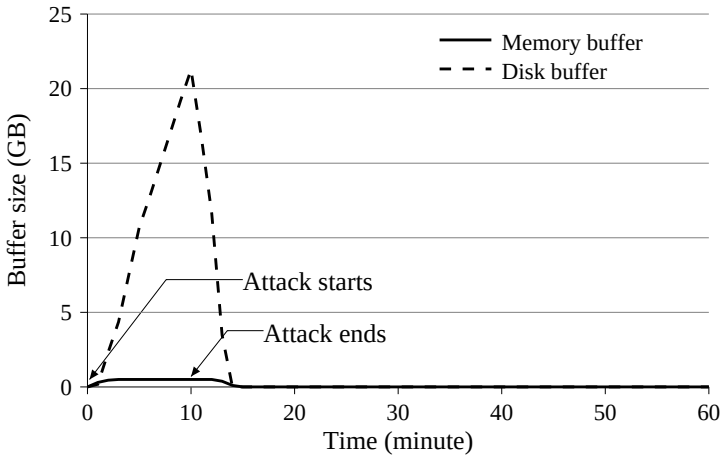


FIGURE 6.9: Size of memory and disk buffers over a 60-minute time period when sending for the first 10 minutes traffic bursts of 1.5 Gbit/s with 30 seconds duration. It takes only 4 minutes for the system to recover from this overload attack.

6.5 Discussion

6.5.1 Real-time Constraints

Our work, much like most of the work in the area of passive network traffic monitoring, focuses on monitoring applications with soft real-time constraints. That is, applications that can afford to receive network packets with some delay, as long as they receive all of them. There exist, however, some traffic monitoring applications that have hard real-time constraints: that is, they can not tolerate any non-trivial delay of any packet. One example of such hard real-time applications are network intrusion *prevention* systems. These systems examine all packets that enter a network before they are allowed to reach their final destination. If a packet is found to be part of an attack then it is dropped, otherwise it is forwarded to its destination. To provide hard-real time guarantees for such applications, their underlying systems are over-provisioned so as to be able to handle the worst-case overload conditions. Our work does not address hard real-time systems, as we do not focus on over-provisioned systems that can absorb the worst case overload in a matter of milliseconds, but we focus on novel approaches, such as two-level memory management and randomized timeouts, which can be implemented on top of ordinary hardware.

6.5.2 Disk Throughput

To be efficient, SPP requires a secondary storage system that is fast enough to write packets at line speed. Fortunately, modern magnetic disks are able to write data at speeds reaching close to half a Gigabit per second, while modern solid state

disks write data an order of magnitude faster than that [4]. If this throughput is not enough to cover a particular line speed, multiple magnetic disks or SSDs can even be used in parallel (e.g., in a RAID array) to achieve ever higher throughput. For example, two SSDs seem to be enough to cover a 10GbE network line. Overall, it seems that a small number of current disks have the bandwidth needed to cover the speeds of current networks. Given the past improvements in disk bandwidth of about 40% per year [94], and the recent breakthroughs in storage technology, we expect commodity storage systems to be able to keep up with network speeds in the years to come at least as easy as, if not easier than, they do today.

6.6 Summary

Under conditions of excessive network or processing load, passive network monitoring applications usually cannot cope with the amount of traffic that needs to be inspected, and the operating system unavoidably drops excess arriving packets. To make matters worse, an attacker may evade detection by intentionally overloading a network intrusion detection system up to the point when it starts dropping packets.

In this chapter we presented Selective Packet Paging, which is based on a two-level memory management approach to buffer (otherwise dropped) packets and tolerate algorithmic complexity attacks, traffic overload attacks, and any other kind of overload conditions for network monitoring and security applications. Empowered with a randomized timeout interval, SPP can detect and isolate algorithmic attack packets, enabling the CPU to be used for more useful purposes, while the application is able to process *all* packets when it recovers from overloads. Selective Packet Paging provides effective packet buffering for several hours; long enough for human (or automatic) intervention to kick in and resolve the overload.

We have implemented SPP within the popular libpcap packet capturing library, so that existing applications can use it without any code modifications. Our experimental evaluation shows that intrusion detection systems, such as Snort, are vulnerable to both algorithmic complexity and traffic overload evasion attacks. Even a few (carefully crafted) packets per second are enough to overload Snort and make it drop the rest of the monitored traffic, missing any subsequent attacks. Using SPP, Snort can handle both algorithmic and traffic overload conditions. Indeed, while Snort on top of the original libpcap missed almost all attacks during an algorithmic overload, SPP enabled Snort to detect 100% of the attacks for speeds of up to 2 Gbit/s.

We believe that as network monitoring applications get more complicated, they will be increasingly vulnerable to algorithmic and traffic overload attacks. SPP offers a memory management approach and a dynamic overload detection technique that provide a seamless solution to this problem without requiring any changes to the monitoring applications themselves.

7

Stream-Oriented Network Traffic Analysis

To make meaningful decisions, many network monitoring applications need to analyze network traffic at the transport layer and above. For instance, NIDSs reconstruct the transport-layer data streams to detect attack vectors spanning multiple packets, and perform traffic normalization to avoid evasion attacks [41, 66, 119]. Similarly, several traffic classification applications are also based on the processing of each TCP-level stream.

Unfortunately, there is a *gap between monitoring applications and underlying traffic capture tools*: Applications increasingly need to reason about higher-level entities and constructs such as TCP flows, HTTP headers, SQL arguments, email messages, and so on, while traffic capture frameworks still operate at the lowest possible level: they provide the raw—possibly duplicate, out-of-order, or overlapping—and in some cases even irrelevant packets that reach the monitoring interface [37,91,92]. Upon receiving the captured packets at user space, monitoring applications usually perform TCP stream reassembly using an existing library such as Libnids [5] or a custom stream reconstruction engine [113, 124]. This results in additional memory copy operations for extracting the payloads of TCP segments and merging them into larger stream "chunks" in contiguous memory. Moreover, it misses several optimization opportunities, such as the early discarding of uninteresting packets before system resources are spent to move them to user level, and assigning different priorities to transport-layer flows so that they can be handled appropriately at lower system layers.

To bridge this gap and address the above concerns, in this chapter we present the *Stream capture library (Scap)*, a unified passive network monitoring framework built around the abstraction of the *Stream*, which is elevated into a first-class object handled by user applications. Designed from the beginning for stream-oriented network monitoring, Scap (i) provides the high-level functionality needed by mon-

itoring applications, and (ii) implements this functionality at the most appropriate place: at user level, at kernel level, or even at the network interface card. On the contrary, existing TCP stream reassembly implementations are confined, by design, to operate at user level and, therefore, are deprived from a rich variety of efficient implementation options.

To enable aggressive optimizations, we introduce the notion of *stream capture*: that is, we elevate the *Stream* into a first-class object that is captured by Scap and handled by user applications. Although previous work treats TCP stream reassembly as a necessary evil [148], used mostly to avoid evasion attacks against intrusion detection and other monitoring systems, we view streams —not packets— as the fundamental abstraction that should be exported to network monitoring applications, and as the right vehicle for the monitoring system to implement aggressive optimizations all the way down to the operating system kernel and network interface card.

To reduce the overhead of unneeded packets, Scap introduces the notion of *subzero packet copy*. Inspired by zero-copy approaches that avoid copying packets from one main memory location to another, Scap not only avoids redundant packet copies, but also avoids bringing some packets in main memory in the first place. We show several cases of applications that are simply not interested in some packets, such as the tails of large flows [24, 86, 89, 107]. Subzero packet copy identifies these packets and does not bring them in main memory at all: they are dropped by the network interface card (NIC) *before* reaching the main memory.

To accommodate heavy loads, Scap introduces the notion of *prioritized packet loss* (PPL). Under heavy load, traditional monitoring systems usually drop arriving packets in a random way, severely affecting any following stream reassembly process. However, these dropped packets and affected streams may be important for the monitoring application, as they may contain an attack or other critical information. Even carefully provisioned systems that are capable of handling full line-rate traffic can be overloaded, e.g., by a sophisticated attacker that sends adversarial traffic to exploit an algorithmic complexity vulnerability and intentionally overload the system [109, 135]. Scap allows applications to (i) define different priorities for different streams and (ii) configure threshold mechanisms that give priority to new and small streams, as opposed to heavy tails of long-running data transfers.

Scap provides a flexible and expressive Application Programming Interface (API) that allows programmers to configure all aspects of the stream capture process, perform complex per-stream processing, and gather per-flow statistics with a few lines of code. Our design introduces two novel features: (i) it enables the early discarding of uninteresting traffic, such as the tails of long-lived connections that belong to large file transfers, and (ii) it offers more control for tolerating packet loss under high load through stream priorities and best-effort reassembly. Scap also avoids the overhead of extra memory copies during the reassembly process by optimally placing TCP segments into stream-specific memory regions, and supports multi-core systems and network adapters with receive-side scaling [75] for transparent parallelization of stream processing.

We have evaluated Scap in a 10GbE environment using real traffic and showed that it outperforms existing alternatives like Libnids [5] and Snort's stream reassembly [124] in a variety of scenarios. For instance, our results demonstrate that Scap can capture and deliver at user level all streams with low CPU utilization for rates up to 5.5 Gbit/s using a single core, while Libnids and Snort start dropping packets at 2.5 Gbit/s due to increased CPU utilization for stream reassembly at user level. A single-threaded Scap pattern matching application can handle 33% higher traffic rates than Snort and Libnids, and can process three times more traffic at 6 Gbit/s. Moreover, the single-threaded Scap pattern matching application can handle traffic speeds of 4 Gbit/s with no loss for stream cutoff values of up to 1MB. In contrast, when Snort and Libnids limit the stream size at user level, even with very low cutoff values, more than 40% of the packets are still dropped at 4 Gbit/s. When eight cores are used for parallel stream processing, Scap can process 5.5 times higher rates with no packet loss.

In summary, the main contributions of this section are:

- We identify a semantic gap: modern network monitoring applications need to operate at the transport layer and beyond, while existing monitoring systems operate at the network layer. To bridge this gap and enable aggressive performance optimizations, we introduce the notion of *stream capture* based on the fundamental abstraction of the *Stream*, which is elevated to a first-class object.

- We introduce *subzero packet copy*, a technique that takes advantage of filtering capabilities of commodity NICs to not only avoid copying uninteresting packets across different memory areas, but to avoid bringing them in main memory altogether.

- We introduce *prioritized packet loss*, a technique that enables graceful adaptation to overload conditions by dropping packets of lower priority streams, and favoring packets that belong to recent and shorter streams.

- We describe the design and implementation of Scap, a framework that incorporates the above features in a kernel-level, multicore-aware subsystem, and provides a flexible and expressive API for building stream-oriented network monitoring applications.

- We experimentally evaluate our implementation and demonstrate that it can capture and deliver transport-layer streams for traffic rates two times higher than previous approaches, while it can also adapt to overload conditions more gracefully and predictably.

The rest of this chapter is organized as follows: in section 7.1 we present the design and basic features of Scap, while in section 7.2 we outline the main Scap

API calls. Then, section 7.3 describes the high-level architecture of Scap, and section 7.4 discusses implementation details. In section 7.5 we experimentally evaluate the performance benefits of Scap, comparing with current network capturing and monitoring libraries for satisfying common monitoring needs, while replaying real network traffic captured in the wild. In section 7.6 we analyze the performance of prioritized packet loss. Finally, section 7.7 compares Scap with other traffic capture frameworks to put our work into context, and section 7.8 summarizes this chapter.

7.1 Design and Features

The design of Scap is driven by two key objectives: programming expressiveness and runtime performance. In this section, we introduce the main aspects of Scap across these two dimensions.

7.1.1 Subzero-Copy Packet Transfer

Several network monitoring applications [24,86,89,107] are interested in analyzing only the first bytes of each connection, especially under high traffic load. This way, they analyze the more useful (for them) part of each stream and discard a significant percentage of the total traffic [89]. For such applications, Scap has incorporated the use of a *cutoff* threshold that truncates streams to a user-specified size, and discards the rest of the stream (and the respective packets) within the OS kernel or even at the NIC, avoiding unnecessary data transfers to user space. Applications can dynamically adjust the cutoff size *per stream*, allowing for greater flexibility.

Besides a stream cutoff size, monitoring applications may be interested in efficiently discarding other types of less interesting traffic. Many applications often use a BPF filter [91] to define which streams they want to process, while discarding the rest. In case of an overload, applications may want to discard traffic from low priority streams or define a stream *overload cutoff* [86, 107]. Also, depending on the stream reassembly mode used by an application, packets belonging to non-established TCP connections or duplicate packets may be discarded. In all such cases, Scap can discard the appropriate packets at an early stage within the kernel, while in many cases packets can be discarded even earlier at the NIC.

To achieve this, Scap capitalizes on modern network interfaces that provide filtering facilities directly in hardware. For example, Intel's 82599 10G interface [74] supports up to 8K perfect match and 32K signature (hash-based) Flow Director filters (FDIR). These filters can be added and removed dynamically, within no more than 10 microseconds, and can match a packet's source and destination IP addresses, source and destination port numbers, protocol, and a flexible 2-byte tuple anywhere within the first 64 bytes of the packet. Packets that match an FDIR filter are directed to the hardware queue specified by the filter. If this hardware queue is not used by the system, the packets will be just dropped at the NIC layer, and they

will never be copied to the system's main memory [39]. When available, Scap uses FDIR filters to implement all above mentioned cases of early packet discarding. Else, the uninteresting packets are dropped within the OS kernel.

7.1.2 Prioritized Packet Loss

Scap introduces *Prioritized Packet Loss* (PPL) to enable the system to invest its resources effectively during overload. This is necessary because sudden traffic bursts or overload conditions may force the packet capturing subsystem to fill up its buffers and randomly drop packets in a haphazard manner. Even worse, attackers may intentionally overload the monitoring system while an attack is in progress so as to evade detection. Previous research in NIDSs has shown that being able to handle different flows [43, 85, 109], or different parts of each flow [86, 107], in different ways can enable the system to invest its resources more effectively and significantly improve detection accuracy. PPL is a priority assignment technique that enables user applications to define the priority of each stream so that in case of overload, packets from low-priority streams are the first ones to go. User applications can also define a threshold for the maximum stream size under overload (*overload_cutoff*). Then, packets situated beyond this threshold are the ones to be dropped.

As long as the percentage of used memory is below a user-defined threshold (called $base\_threshold$), PPL drops no packets. When, however, the used memory exceeds the $base\_threshold$, PPL kicks in: it first divides the memory above $base\_threshold$ into n (equal to the number of used priorities) regions using $n + 1$ equally spaced watermarks (i.e., $watermark_0$, $watermark_1$, ..., $watermark_n$), where $watermark_0 = base\_threshold$ and $watermark_n = memory\_size$. When a packet belonging to a stream with the i_{th} priority level arrives, PPL checks the percentage of memory used by Scap at that time. If it is above $watermark_i$, the packet is dropped. Otherwise, if the percentage of memory used is between $watermark_i$ and $watermark_{i-1}$, PPL makes use of the *overload_cutoff*, if it has been defined by the user. Then, if the packet is located in its stream beyond the *overload_cutoff* byte, it is dropped. In this way, high priority streams, as well as newly created and short streams if an *overload_cutoff* is defined, will be accommodated with higher probability.

7.1.3 Flexible Stream Reassembly

To support monitoring at the transport layer, Scap provides different modes of TCP stream reassembly. The two main objectives of stream reassembly in Scap are: (i) to provide transport-layer reassembled chunks in continuous memory regions, which facilitates stream processing operations, and (ii) to perform protocol normalization [66, 150]. Scap currently supports two different modes of TCP stream reassembly: `SCAP_TCP_STRICT` and `SCAP_TCP_FAST`. In the strict mode, streams are reassembled according to existing guidelines [41, 150], offering protection

against evasion attempts based on IP/TCP fragmentation. In the fast mode, streams are reassembled in a *best-effort* way, offering resilience against packet loss caused in case of overloads. In this mode, Scap follows the semantics of the strict mode as closely as possible, e.g., by handling TCP retransmissions, out-of-order packets, and overlapping segments. However, to accommodate for lost segments, stream data is written without waiting for the correct next sequence number to arrive. In that case, Scap sets a flag to report that errors occurred during the reassembly of a particular chunk.

Scap uses target-based stream reassembly to implement different TCP reassembly policies according to different operating systems. Scap applications can set a different reassembly policy per each stream. This is motivated by previous work, which has shown that stream reassembly performed in a NIDS may not be accurate [119]. For instance, the reconstructed data stream may differ from the actual data stream observed by the destination. This is due to the different TCP reassembly policies implemented by different operating systems, e.g., when handling overlapping segments. Thus, an attacker can exploit such differences to evade detection. Shankar and Paxson [131] developed an active mapping solution to determine what reassembly policy a NIDS should follow for each stream. Similarly to Scap, Snort uses target-based stream reassembly [103] to define the reassembly policy per host or subnet.

Scap also supports UDP: a UDP stream is the concatenation of the payloads of the arriving packets of the respective UDP flow. For other protocols without sequenced delivery, Scap return each packet for processing without reassembly.

7.1.4 Parallel Processing and Locality

Scap has inherent support for multi-core systems, hiding from the programmer the complexity of creating and managing multiple processes or threads. This is achieved by transparently creating a number of worker threads for user-level stream processing (typically) equal to the number of the available cores. Using affinity calls, the mapping of threads to CPU cores is practically one-to-one. Scap also dedicates a kernel thread on each core for handling packet reception and stream reassembly. The kernel and worker threads running on the same core process the same streams. As each stream is assigned to only one kernel and worker thread, all processing of a particular stream is done on the same core, reducing, in this way, context switches, cache misses [42, 115], and inter-thread synchronization operations. The kernel and worker threads on each core communicate through shared memory and events: a new event for a stream is created by the kernel thread and is handled by the worker thread using a user-defined callback function for stream processing.

To balance the network traffic load across multiple NIC queues and cores, Scap uses both static hash-based approaches, such as Receive Side Scaling (RSS) [75], and dynamic load balancing approaches, such as flow director filters (FDIR) [74]. This provides resiliency to short-term load imbalance that could adversely affect

application performance. First, Scap detects a load imbalance when one of the cores is assigned a portion of the total streams larger than a threshold. Then, subsequent streams assigned by RSS to this core are re-directed with an FDIR to the core that handles the lowest number of streams at the time.

7.1.5 Performance Optimizations

In case that multiple applications running on the same host monitor the same traffic, Scap provides all of them with a shared copy of each stream. Thus, the stream reassembly operation is performed only once within the kernel, instead of multiple times for each user-level application. If applications have different configurations, e.g., for stream size cutoff or BPF filters, the capture system takes a best effort approach to satisfy all requirements. For instance, it sets the largest among the cutoff sizes for all streams, and keeps streams that match at least one of the filters, marking the applications that should receive each stream and their respective cutoff.

Performing stream reassembly in the kernel also offers significant advantages in terms of cache locality. Existing user-level TCP stream reassembly implementations receive packets of different flows highly interleaved, which results in poor cache locality [111]. In contrast, Scap provides user-level applications with reassembled streams instead of randomly interleaved packets, allowing for improved memory locality and reduced cache misses.

7.2 Scap API

Scap is based around the abstraction of the *stream*: a reconstructed TCP session between two endpoints defined by a 5-tuple (protocol, source and destination IP address, source and destination port). Monitoring applications receive a unique stream descriptor `stream_t` for each new stream. This descriptor can be used to access all information, data, and statistics about the stream, and is provided as a parameter to all stream manipulation functions. Table 7.1 presents the main fields of the `stream_t` structure, and Table 7.2 lists the main functions of the Scap API. In the following, we give a brief overview of the main functions of the API, and provide two simple examples that demonstrate its expressiveness and flexibility.

7.2.1 Initialization

An Scap program begins with the creation of an Scap socket using `scap_create()`, which specifies the interface to be monitored. The programmer can also specify various properties, such as the memory size of the buffer for storing stream data, the stream reassembly mode, and whether the application needs to receive the individual packets of each stream. Upon successful creation, the returned `scap_t` descriptor, which keeps configuration parameters, is used for all subsequent configuration operations. These include setting a BPF filter [91] to receive a subset of

Data field	Description
stream_hdr ***hdr;***	*Stream header*
uint32_t **src_ip**, **dst_ip**;	Source/Destination IP address
uint16_t **src_port**, **dst_port**;	Source/Destination port
uint8_t **protocol**;	Protocol
uint8_t **direction**;	Stream direction
stream_stats ***stats;***	*Stream statistics*
struct timeval **start**, **end**;	Beggining/end time
uint64_t **bytes**,	Total bytes,
bytes_dropped,	dropped bytes,
bytes_discarded,	discarded bytes,
bytes_captured;	captured bytes
uint32_t **pkts**,	Total packets,
pkts_dropped,	dropped packets,
pkts_discarded,	discarded packets,
pkts_captured;	captured packets
	Other fields
uint8_t **status**;	Stream status
uint8_t **error**;	Error flags
char *__data__;	Pointer to last chunk's data
int **data_len**;	Data length of the last chunk
stream_t *__opposite__;	Stream in the opposite direction
int **cutoff**;	Stream's cutoff
int **priority**;	Stream's priority
int **chunk_size**;	Stream's chunk size
int **chunks**;	Stream's total chunks
int **processing_time**;	Stream's processing time

TABLE 7.1: Data fields of the stream descriptor `stream_t`.

the traffic, cutoff values for different stream classes or stream directions, the number of worker threads for balancing stream processing among the available cores, the chunk size, the overlap size between subsequent chunks, and an optional timeout for delivering the next chunk for processing. The `overlap` argument is used when some of the last bytes of the previous chunk are also needed in the beginning of the next chunk, e.g., for matching a pattern that might span consecutive chunks. The `flush_timeout` parameter can be used to deliver for processing a chunk smaller than the chunk size when this timeout passes, in case the user needs to ensure timely processing.

7.2.2 Stream Processing

Scap allows programmers to write and register callback functions for three different types of events: stream creation, the availability of new stream data, and stream termination. When a stream is created or terminated, or when enough data have been captured for a stream's chunk processing, a new event is triggered and the respective callback is executed. Each callback function takes as a single argument a `stream_t` descriptor `sd`, which corresponds to the stream that triggered the event. As shown in Table 7.1, this descriptor provides access to detailed information about the stream, such as the stream's IP addresses, port numbers, protocol, and direc-

Scap Function Prototype	Description
`scap_t *scap_create(const char *device, int memory_size, int reassembly_mode, int need_pkts)`	Creates an Scap socket
`int scap_set_filter(scap_t *sc, char *bpf_filter)`	Applies a BPF filter to an Scap socket
`int scap_set_cutoff(scap_t *sc, int cutoff)`	Changes the default stream cutoff value
`int scap_add_cutoff_direction(scap_t *sc, int cutoff, int direction)`	Sets a different cutoff value for each direction
`int scap_add_cutoff_class(scap_t *sc, int cutoff, char* bpf_filter)`	Sets a different cutoff value for a subset of the traffic
`int scap_set_worker_threads(scap_t *sc, int thread_num)`	Sets the number of threads for stream processing
`int scap_set_parameter(scap_t *sc, int parameter, int value)`	Changes defaults: inactivity_timeout, chunk_size, overlap_size, flush_timeout, base_threshold, overload_cutoff
`int scap_dispatch_creation(scap_t *sc, void (*handler)(stream_t *sd))`	Registers a callback routine for handling stream creation events
`int scap_dispatch_data(scap_t *sc, void (*handler)(stream_t *sd))`	Registers a callback routine for processing newly arriving stream data
`int scap_dispatch_termination(scap_t *sc, void (*handler)(stream_t *sd))`	Registers a callback routine for handling stream termination events
`int scap_start_capture(scap_t *sc)`	Begins stream processing
`void scap_discard_stream(scap_t *sc, stream_t *sd)`	Discards the rest of a stream's traffic
`int scap_set_stream_cutoff(scap_t *sc, stream_t sd, int cutoff)`	Sets the cutoff value of a stream
`int scap_set_stream_priority(scap_t *sc, stream_t *sd, int priority)`	Sets the priority of a stream
`int scap_set_stream_parameter(scap_t *sc, stream_t *sd, int parameter, int value)`	Sets a stream's parameter: inactivity_timeout, chunk_size, overlap_size, flush_timeout, reassembly_mode
`int scap_keep_stream_chunk(scap_t *sc, stream_t *sd)`	Keeps the last chunk of a stream in memory
`char *scap_next_stream_packet(stream_t *sd, struct scap_pkthdr *h)`	Returns the next packet of a stream
`int scap_get_stats(scap_t *sc, scap_stats_t *stats)`	Returns the next packet of a stream
`void scap_close(scap_t *sc)`	Reads overall statistics for all streams
	Closes an Scap socket

TABLE 7.2: The main functions of the Scap API.

tion, as well as useful statistics such as byte and packet counters for all, dropped, discarded, and captured packets, and the timestamps of the first and last packet of the stream. Among the rest of the fields, the `sd->status` field indicates whether the stream is active or closed (by TCP FIN/RST or by inactivity timeout), or if its stream cutoff has been exceeded, and the `sd->error` field indicates stream reassembly errors, such as incomplete TCP handshake or invalid sequence numbers. There is also a pointer to the `stream_t` in the opposite direction, and stream's properties like cutoff, priority, and chunk size.

The stream processing callback can access the last chunk's data and its size through the `sd->data` and `sd->data_len` fields. In case no more data is needed, `scap_discard_stream()` can notify the Scap core to stop collecting data for this stream. Chunks can be efficiently merged with following ones using `scap_ keep_chunk()`. In the next invocation, the callback will receive a larger chunk consisting of both the previous and the new one. Using the stream descriptor, the application is able to set the stream's priority, cutoff, and other parameters like stream's chunk size, overlap size, flush timeout, and reassembly mode.

In case they are needed by an application, individual packets can be delivered using `scap_next_stream_packet()`. Packet delivery is based on the chunk's

data and metadata kept by Scap's packet capture subsystem for each packet. Based on this metadata, even reordered, duplicate, or packets with overlapping sequence numbers can be delivered in the same order as captured. This allows Scap to support packet-based processing along with stream-based processing, e.g., to allow the detection of TCP attacks such as ACK splitting [126]. The only difference between Scap's packet delivery and packet-based capturing systems is that packets from the same stream are processed together, due to the chunk-based delivery. As an added benefit, such flow-based packet reordering has been found to significantly improve cache locality [111].

The stream's processing time and the total number of processed chunks are available through the `sd->processing_time` and `sd->chunks` fields. This enables the identification of streams that are processed with very slow rates and delay the application, e.g., due to algorithmic complexity attacks [109, 135]. Upon the detection of such a stream, the application can handle it appropriately, e.g., by discarding it or reducing its priority, to ensure that this adversarial traffic will not affect the application's correct operation.

7.2.3 Use Cases

We now show two simple applications written with Scap.

Flow-Based Statistics Export

The following listing shows the code of an Scap application for gathering and exporting per-flow statistics. Scap already gathers these statistics and stores them in the `stream_t` structure of each stream, so there is no need to receive any stream data. Thus, the stream cutoff can be set to zero, to efficiently discard all data. All the required statistics for each stream can be retrieved upon stream termination by registering a callback function.

```
1  scap_t *sc = scap_create("eth0", SCAP_DEFAULT,
2      SCAP_TCP_FAST, 0);
3  scap_set_cutoff(sc, 0);
4  scap_dispatch_termination(sc, stream_close);
5  scap_start_capture(sc);
6
7  void stream_close(stream_t *sd) {
8    export(sd->hdr.src_ip, sd->hdr.dst_ip,
9      sd->hdr.src_port, sd->hdr.dst_port,
10     sd->stats.bytes, sd->stats.pkts,
11     sd->stats.start, sd->stats.end);
12 }
```

In line 1 we create a new Scap socket for capturing streams from the `eth0` interface. Then, we set the stream cutoff to zero (line 3) for discarding all stream data,

we set the `stream_close()` as a callback function to be called upon stream termination (line 4), and finally we start the capturing process (line 5). The `stream_close()` function (lines 7–12) exports the statistics of the stream through the `sd` descriptor that is passed as its argument.

Pattern Matching

The following listing shows the few lines of code that are required using Scap for an application that searches for a set of known patterns in the captured reassembled TCP streams.

```
1 scap_t *sc = scap_create("eth0", 512M,
2     SCAP_TCP_FAST, 0);
3 scap_set_worker_threads(sc, 8);
4 scap_dispatch_data(sc, stream_process);
5 scap_start_capture(sc);
6
7 void stream_process(stream_t *sd) {
8  search(patterns, sd->data, sd->len, MatchFound);
9 }
```

We begin by creating an Scap socket without setting a cutoff, so that all traffic is captured and processed (lines 1–2). Then, we configure Scap with eight worker threads, each pinned to a single CPU core (assuming a machine with eight cores), to speed up pattern matching with parallel stream processing. Finally, we register `stream_process()` as the callback function for processing stream chunks (line 4) and start the capturing process (line 5). The `search()` function looks for the set of known patterns within `sd->len` bytes starting from the `sd->data` pointer, and calls the `MatchFound()` function in case of a match.

7.3 Architecture

This section describes the architecture of the Scap monitoring framework for stream-oriented network traffic capturing and processing.

7.3.1 Kernel-level and User-level Support

Scap consists of two main components: a loadable kernel module and a user-level API stub, as shown in Figure 7.1. Applications communicate through the Scap API stub with the kernel module to configure the capture process and receive monitoring data. Configuration parameters are passed to the kernel through the Scap socket interface. Accesses to `stream_t` records, events, and actual stream data are handled through shared memory. For user-level stream processing, the stub receives events from the kernel module and calls the respective callback function for each event.

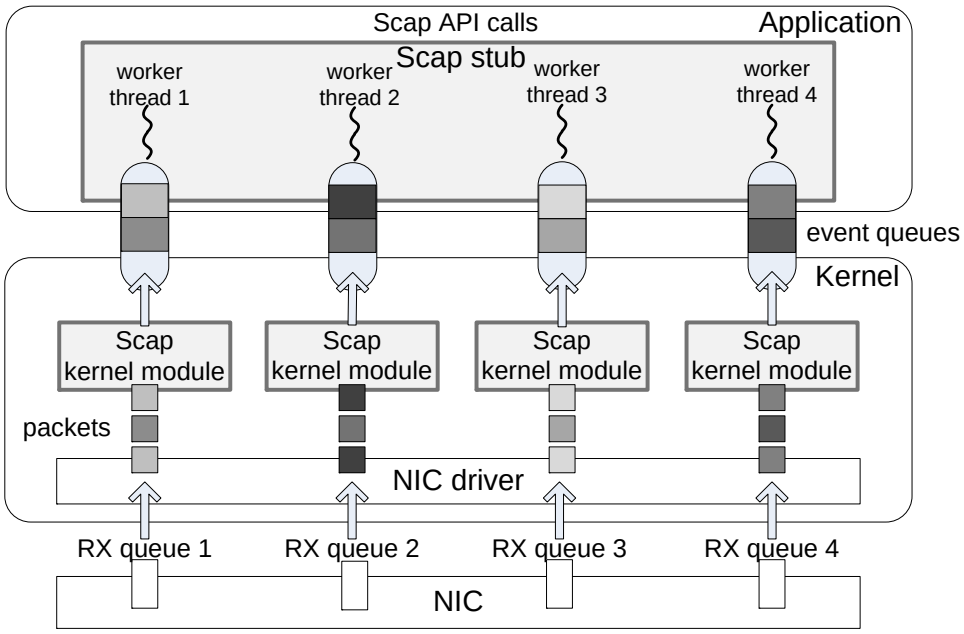


FIGURE 7.1: Overview of Scap's architecture.

The overall operation of the Scap kernel module is depicted in Figure 7.2. Its core is a software interrupt handler that receives packets from the network device. For each packet, it locates the respective `stream_t` record through a hash table and updates all relevant fields (`stream_t` handling). If a packet belongs to a new stream, a new `stream_t` record is created and added into the hash table. Then, it extracts the actual data from each TCP segment, by removing the protocol headers, and stores it in the appropriate memory page, depending on the stream in which it belongs (memory management). Whenever a new stream is created or terminated, or a sufficient amount of data has been gathered, the kernel module generates a respective event and enqueues it to an event queue (event creation).

7.3.2 Parallel Packet and Stream Processing

To scale performance, Scap uses all available cores in the system. To efficiently utilize multi-core architectures, modern network interfaces can distribute incoming packets into multiple hardware receive queues. To balance the network traffic load across the available queues and cores, Scap uses both RSS [75], which uses a hash function based on the packets' 5-tuple, and dynamic load balancing, using flow director filters [74], to deal with short-term load imbalance. To map the two different streams of each bi-directional TCP connection to the same core, we modify the RSS seeds as proposed by Woo and Park [151].

Each core runs a separate instance of the NIC driver and Scap kernel module to handle interrupts and packets from the respective hardware queue. Thus, each Scap

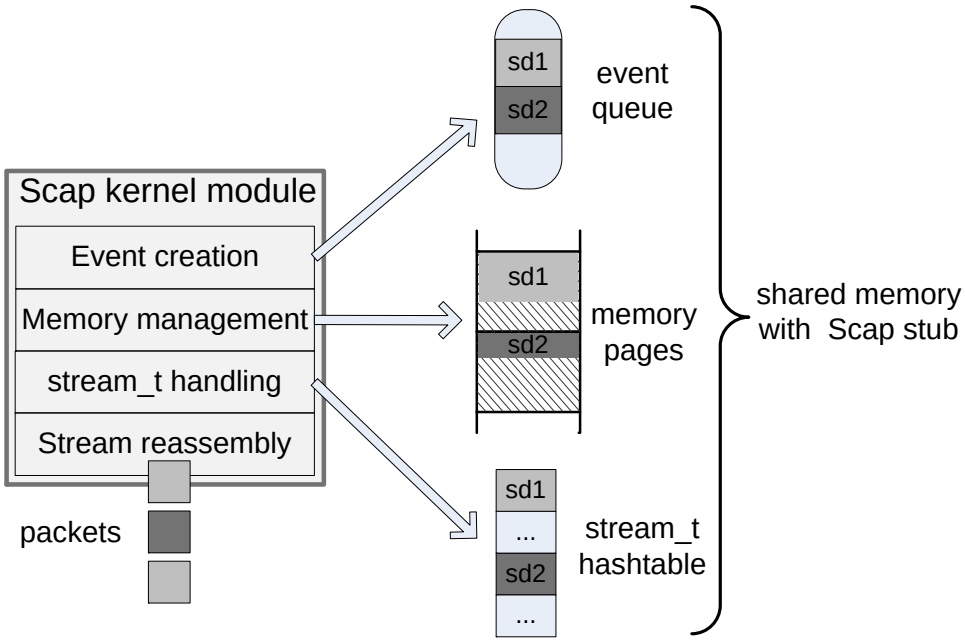


FIGURE 7.2: The operation of the Scap kernel module.

instance running on each core will receive a different subset of network streams, as shown in Figure 7.1. Consequently, the stream reassembly process is distributed across all the available cores. To match the level of parallelism provided by the Scap kernel module, the Scap's user-level stub creates as many worker threads as the available cores, hiding from the programmer the complexity of creating and managing multiple processes or threads. Each worker thread processes the streams delivered to its core by its kernel-level counterpart. This collocation of user-level and kernel-level threads that work on the same data maximizes locality of reference and cache affinity, reducing, in this way, context switches, cache misses [42, 115], and inter-thread synchronization. Each worker thread polls a separate event queue for events created by the kernel Scap thread running on the same core, and calls the respective callback function registered by the application to process each event.

7.4 Implementation

We now give more details on the implementation of the Scap monitoring framework.

7.4.1 Scap Kernel Module

The Scap kernel module implements a new network protocol for receiving packets from network devices, and a new socket class, PF_SCAP, for communication between the Scap stub and the kernel module. Packets are transferred to memory

through DMA, and the driver schedules them for processing within the software interrupt handler—the Scap's protocol handler in our case.

7.4.2 Fast TCP Reassembly

For each packet, the Scap kernel module finds and updates its respective `stream_t` record, or creates a new one. For fast lookup, we use a hash table by randomly choosing a hash function during initialization. Based on the transport-layer protocol headers, Scap extracts the packet's data and writes them directly to the current memory offset indicated in the `stream_t` record. Packets belonging to streams that exceed their cutoff value, as well as duplicate or overlapping TCP segments, are discarded immediately without unnecessarily spending further CPU and memory resources for them. Streams can expire explicitly (e.g., via TCP FIN/RST), or implicitly, due to an inactivity timeout. For the latter, Scap maintains an *access list* with the active streams sorted by their last access time. Upon packet reception, the respective `stream_t` record is simply placed at the beginning of the access list, to keep it sorted. Periodically, starting from the end of the list, the kernel module compares the last access time of each stream with the current time, and expires all streams for which no packet was received within the specified period by creating stream termination events.

7.4.3 Memory Management

Reassembled streams are stored in a large memory buffer allocated by the kernel module and mapped in user level by the Scap stub. The size of this buffer is given as argument in the `scap_create()` function (`buffer_len`). The kernel module allocates the respective memory pages during initialization, and it is responsible to manage the usage of this memory among the several streams. For each stream, a contiguous memory block is allocated (by our own memory allocator) according to the stream's chunk size. When this block fills up, the chunk is delivered for processing (by creating a respective event) and a new block is allocated for the next chunk. The Scap stub has access to this block through memory mapping, so an offset is enough for locating each stored chunk.

To avoid dynamic allocation overhead, a large number of `stream_t` records are pre-allocated during initialization, and are memory-mapped by the Scap stub. More records are allocated dynamically as needed. Thus, the number of streams that can be tracked concurrently is not limited by Scap.

7.4.4 Event Creation

A new event is triggered on stream creation, stream termination, and whenever stream data is available for processing. A data event can be triggered for one of the following reasons: (i) a memory chunk fills up, (ii) a flush timeout is passed, (iii) a cutoff value is exceeded, or (iv) a stream is terminated. When a stream's

cutoff threshold is reached, Scap creates a final data processing event for its last chunk. However, its `stream_t` record remains in the hash table and in the access list, so that monitoring continues throughout its whole lifetime. This is required for gathering flow statistics and generating the appropriate termination event.

To avoid contention when the Scap kernel module runs in parallel across several cores, each core inserts events in a separate queue. When a new event is added into a queue, the `sk_data_ready()` function is called to wake up the corresponding worker thread, which calls `poll()` whenever its event queue is empty. Along with each event, the Scap stub receives and forwards to the user-level application a pointer to the respective `stream_t` record. To avoid race conditions between the Scap kernel module and the application, Scap maintains a second instance of each `stream_t` record. The first copy is updated within the kernel, while the second is read by the user-level application. The kernel module updates the necessary fields of the second `stream_t` instance right before a new event for this stream is enqueued.

7.4.5 Hardware Filters

Packets taking part in the TCP three-way handshake are always captured. When the cutoff threshold is triggered for a stream, Scap adds dynamically the necessary FDIR filters to drop at the NIC layer all subsequent packets belonging to this stream. Note that although packets are dropped before they reach main memory, Scap needs to know when a stream ends. For this reason, we add filters to drop only packets that contain actual data segments (or TCP acknowledgements), and still allow Scap to receive TCP RST or FIN packets that may terminate a stream.

This is achieved using the flexible 2-byte tuple option of FDIR filters. We have modified the NIC driver to allow for matching the offset, reserved, and TCP flags 2-byte tuple in the TCP header. Using this option, we add two filters for each stream: the first matches and drops TCP packets for which only the ACK flag is set, and the second matches and drops TCP packets for which only the ACK and PSH flags are set. The rest of the filter fields are based on each stream's 5-tuple. Thus, only TCP packets with RST or FIN flag will be forwarded to Scap kernel module for stream termination.

Streams may also be terminated based on an inactivity timeout. For this reason Scap associates a timeout with each filter installed to the NIC, and removes the filter when this timeout expires. To remove filters, Scap keeps a list with all filters sorted based on their timeout values. Thus, an FDIR filter is removed (i) when a TCP RST or FIN packet arrives for a given stream, or (ii) when the timeout associated with a filter expires. Note that in the second case the stream may still be active, so if a packet of this stream arrives upon the removal of its filter, Scap will immediately re-install the filter. This is because the cutoff of this stream has exceeded and the stream is still active. To handle long running streams, re-installed filters get a timeout twice as large as before. In this way, long-running flows will only be evicted a logarithmic number of times from NIC's filters. If there is no

space left on the NIC to accommodate a new filter, a filter with a small timeout is evicted, as it does not correspond to a long-lived stream.

Scap needs to provide accurate flow statistics upon the termination of streams that had exceeded their cutoff, even if most of their packets were discarded at the NIC. Unfortunately, existing NICs provide only aggregate statistics for packets across all filters—not per each filter. However, Scap is able to estimate accurate per-flow statistics, such as flow size and flow duration for TCP streams, based on the TCP sequence numbers of the RST/FIN packets. Also, by removing the NIC filters when their timeout expires, Scap receives packets from these streams periodically and updates their statistics.

Our implementation is based on the Intel 82599 NIC [74], which supports RSS and flow director filters. Similarly to this card, most modern 10GbE NICs such as Solarflare [138], SMC [134], Chelsio [26], and Myricom [101], also support RSS and filtering capabilities, so Scap can be effectively used with these NICs as well.

7.4.6 Handling Multiple Applications

Multiple applications can use Scap concurrently on the same machine. Given that monitoring applications require only read access to the stream data, there is room for stream sharing to avoid multiple copies and improve overall performance. To this end, all Scap sockets share a single memory buffer for stream data and `stream_t` records. As applications have different requirements, Scap tries to combine and generalize all requirements at kernel level, and apply application-specific configurations at user level.

7.4.7 Packet Delivery

An application may be interested in receiving both reassembled streams, as well as their individual packets, e.g., to detect TCP-level attacks [126]. Scap supports the delivery of the original packets as captured from the network, if an application indicates that it needs them. Then, Scap internally uses another memory-mapped buffer that contains records for each packet of a stream. Each record contains a packet header with the timestamp and capture length, and a pointer to the original packet payload in the stream.

7.4.8 API Stub

The Scap API stub uses `setsockopt()` to pass parameters to kernel module for handling API calls. When `scap_start_capture()` is called, each worker thread runs an event-dispatch loop that polls its corresponding event queue, reads the next available event, and executes the registered callback function for this event. The event queues contain `stream_t` objects, which have an `event` field and a pointer to the next `stream_t` in the event queue. If this pointer is NULL, then there is no event in the queue, and the stub calls `poll()` to wait for future events.

7.5 Experimental Evaluation

We experimentally evaluate the performance of Scap, comparing it to other stream reassembly libraries for common monitoring tasks, such as flow statistics export and pattern matching, while replaying real network traffic at different rates.

7.5.1 Experimental Environment

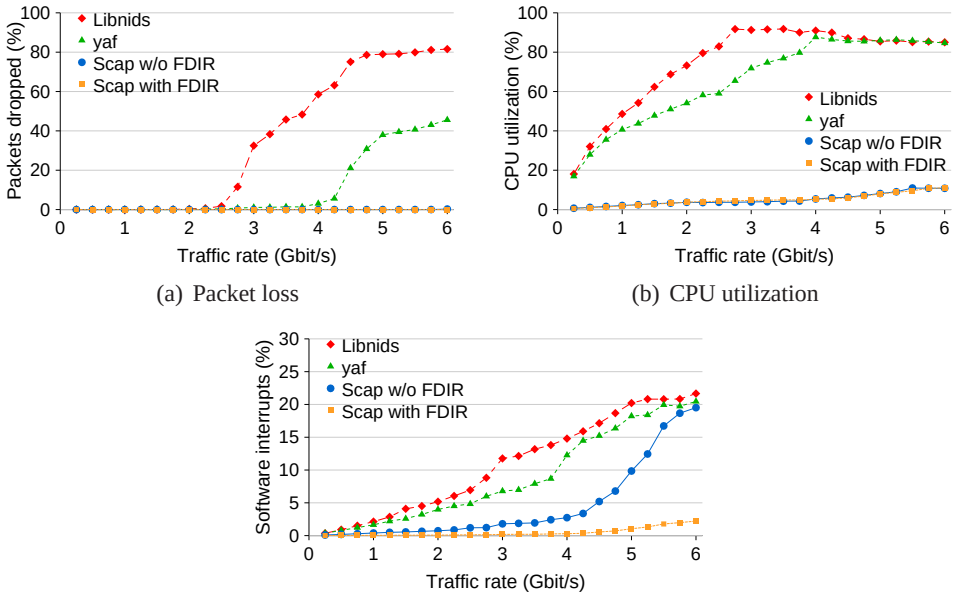
The hardware We use a testbed comprising two PCs interconnected through a 10 GbE switch. The first, equipped with two dual-core Intel Xeon 2.66 GHz CPUs with 4MB L2 cache, 4GB RAM, and an Intel 82599EB 10GbE NIC, is used for traffic generation. The second, used as a monitoring sensor, is equipped with two quad-core Intel Xeon 2.00 GHz CPUs with 6MB L2 cache, 4GB RAM, and an Intel 82599EB 10GbE NIC used for stream capture. Both PCs run 64-bit Ubuntu Linux (kernel version 2.6.32).

The trace To evaluate stream reassembly implementations with real traffic, we replay a one-hour long anonymized trace captured at the access link that connects to the Internet a University campus with thousands of hosts. The trace contains 58,714,906 packets and 1,493,032 flows, totaling more than 46GB, 95.4% of which is TCP traffic. To achieve high replay rates (up to 6 Gbit/s) we split the trace in smaller parts of 1GB that fit into main memory, and replay each part 10 times while the next part is being loaded in memory.

The parameters We compare the following systems: (i) Scap, (ii) Libnids v1.24 [5], (iii) YAF v2.1.1 [73], a libpcap-based flow export tool, and (iv) the Stream5 preprocessor of Snort v2.8.3.2 [124]. YAF, Libnids and Snort rely on libpcap [92], which uses the PF_PACKET socket for packet capture on Linux. Similarly to Scap's kernel module, the PF_PACKET kernel module runs as a software interrupt handler that stores incoming packets to a memory-mapped buffer, shared with libpcap's user-level stub. In our experiments, the size of this buffer is set to 512MB, and the buffer size for reassembled streams is set to 1GB for Scap, Libnids, and Snort. We use a chunk size of 16KB, the SCAP_TCP_FAST reassembly mode, and an inactivity timeout of 10 seconds. The majority of TCP streams terminate explicitly with TCP FIN or RST packet, but we also use an inactivity timeout to expire UDP, and TCP flows that do not close normally. As we replay the trace at higher rates than its actual capture rate, an inactivity timeout of 10 seconds is a reasonable choice.

7.5.2 Flow-Based Statistics Export: Drop Anything Not Needed

In our first experiment we evaluate the performance of Scap for exporting flow statistics, comparing with YAF and with a Libnids-based program that receives reassembled flows. By setting the stream cutoff value to zero, Scap discards all stream data after updating stream statistics. When Scap is configured to use the



(a) Packet loss

(b) CPU utilization

(c) Software interrupt load

FIGURE 7.3: Performance comparison of flow-based statistics export for YAF, Libnids, and Scap, for varying traffic rates.

FDIR filters, the NIC discards all packets of a flow after TCP connection establishment, except from the TCP FIN/RST packets, which are used by Scap for flow termination. Although Scap can use all eight available cores, for a fair comparison, we configure it to use a single worker thread, as YAF and Libnids are single-threaded. However, for all tools, interrupt handling for packet processing in the kernel takes advantage of all cores, utilizing NIC's multiple queues.

Figures 7.3(a), 7.3(b), and 7.3(c) present the percentage of dropped packets, the average CPU utilization of the monitoring application on a single core, and the software interrupt load while varying the traffic rate from 250 Mbit/s to 6 Gbit/s. We see that Libnids starts losing packets when the traffic rate exceeds 2 Gbit/s. The reason can be seen in Figures 7.3(b) and 7.3(c), where the total CPU utilization of Libnids exceeds 90% at 2.5 Gbit/s. YAF performs slightly better than Libnids, but when the traffic reaches 4 Gbit/s, it also drives CPU utilization to 100% and starts losing packets as well. This is because both YAF and Libnids receive all packets in user space and then drop them, as the packets themselves are not needed by the monitoring application.

Scap processes all packets even at 6 Gbit/s load. As shown in Figure 7.3(b), the CPU utilization of the Scap application is always less than 10%, as it practically does not do any work at all. All the work has already been done by Scap's kernel module. One would expect the overhead of this module (shown in Figure 7.3(c)) to be relatively high. Fortunately, however, the software interrupt load of Scap is even

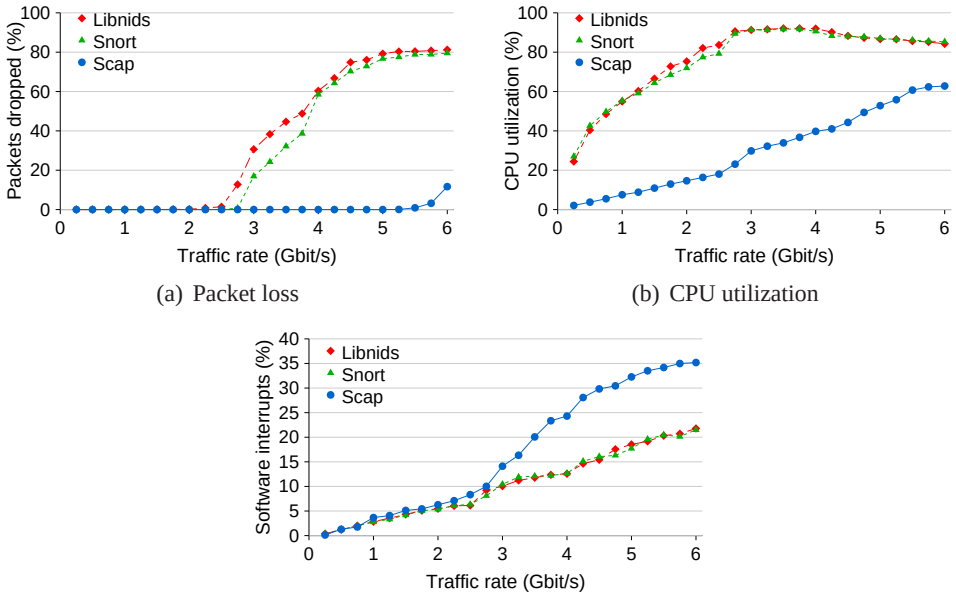
lower compared to YAF and Libnids, even when FDIR filters are not used, because Scap does not copy the incoming packets around: as soon as a packet arrives, the kernel module accesses only the needed information from its headers, updates the respective *stream_t*, and just drops it. In contrast, Libnids and YAF receive all packets to user space, resulting in much higher overhead. YAF performs better than Libnids because it receives only the first 96 bytes of each packet and it does not perform stream reassembly.

When Scap uses FDIR filters to discard the majority of the packets at NIC layer it achieves even better performance. Figure 7.3(c) shows that the software interrupt load is significantly lower with FDIR filters: as little as 2% for 6 Gbit/s. Indeed, Scap with FDIR brings into main memory as little as 3% of the total packets—just the packets involved in TCP session creation and termination. The rest of the packets are just not needed, and they are never brought in the main memory.

7.5.3 Delivering Streams to User Level: The Cost of an Extra Memory Copy

In this experiment, we explore the performance of Scap, Snort, and Libnids when delivering reassembled streams to user level without any further processing. The Scap application receives all data from all streams with no cutoff, and runs as a single thread. Snort is configured with only the Stream5 preprocessor enabled, without any rules. The Libnids application also receives all the reassembled TCP and UDP streams, without any operation of them. Figure 7.4(a) shows the percentage of dropped packets as a function of the traffic rate. Scap delivers all steams without any packet loss for rates up to 5.5 Gbit/s. On the other hand, Libnids starts dropping packets at 2.5 Gbit/s (drop rate: 1.4%) and Snort at 2.75 Gbit/s (drop rate: 0.7%). Thus, Scap is able to deliver reassembled streams to the monitoring applications for more than two times higher traffic rates. When the input traffic reaches 6 Gbit/s, Libnids drops 81.2% and Snort 79.5% of the total packets received.

The reason for this performance difference lies in the extra memory copy operations needed for stream reassembly at user level. When a packet arrives for Libnids and Snort, the kernel writes it in the next available location in a common ring buffer. When performing stream reassembly, Libnids and Snort may have to *copy* each packet's payload from the ring buffer to a memory buffer allocated specifically for this packet's stream. Scap avoids this extra copy operation because the kernel module copies the packet's data *not* to a common buffer, but directly to a memory buffer allocated specifically for this packet's stream. Figure 7.4(b) shows that the CPU utilization of the Scap user-level application is considerably lower than the utilization of Libnids and Snort, which at 3 Gbit/s exceeds 90%, saturating the processor. In contrast, the CPU utilization for the Scap application is less then 60% even for speeds up to 6 Gbit/s, as the user application does very little work: all the stream reassembly is performed in the kernel module, which increases the software interrupt load, as can be seen in Figure 7.4(c).



(a) Packet loss

(b) CPU utilization

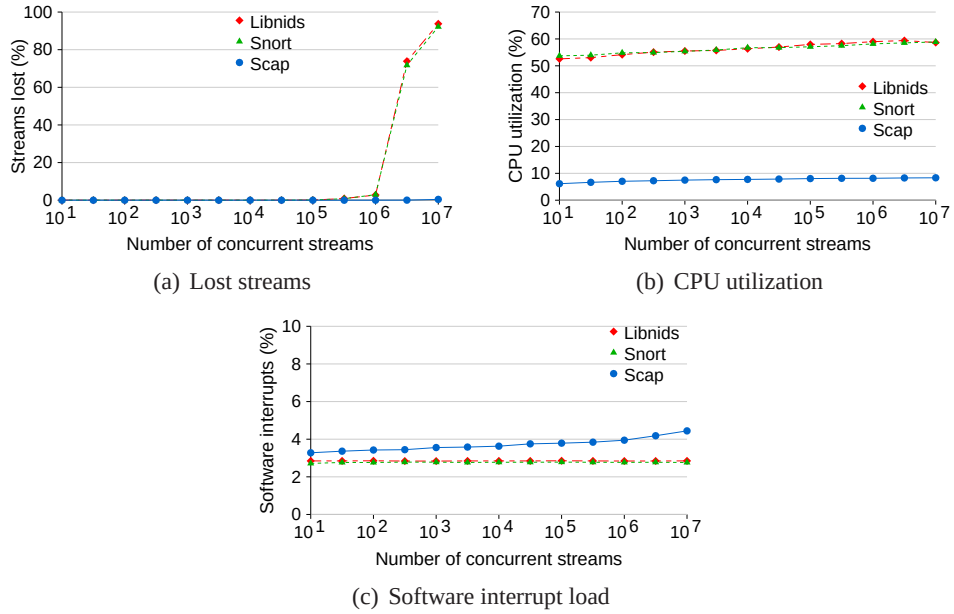
(c) Software interrupt load

FIGURE 7.4: Performance comparison of stream delivery for Snort, Libnids, and Scap, for varying traffic rates.

7.5.4 Concurrent Streams

An attacker could try to saturate the flow table of a stream reassembly library by creating a large number of established TCP flows, so that a subsequent malicious flow cannot be stored. In this experiment, we evaluate the ability of Scap, Libnids, and Snort to handle such cases while increasing the number of concurrent TCP streams up to 10 million. Each stream consists of 100 packets with the maximum TCP payload, and streams are multiplexed so that the desirable number of concurrent streams is achieved. For each case, we create a respective packet trace and then replay it at a constant rate of 1 Gbit/s, as we want to evaluate the effect of concurrent streams without increasing the traffic rate. As in the previous experiment, the application uses a single thread and receives all streams at user level, without performing any further processing.

Figure 7.5 shows that Scap scales well with the number of concurrent streams: as we see in Figure 7.5(a), no stream is lost even for 10 million concurrent TCP streams. Also, Figures 7.5(b) and 7.5(c) show that the CPU utilization and software interrupt load of Scap slightly increase with the number of concurrent streams, as the traffic rate remains constant. On the other hand, Snort and Libnids cannot handle more than one million concurrent streams, even though they can handle 1 Gbit/s traffic with less than 60% CPU utilization. This is due to internal limits that these libraries have for the number of flows they can store in their data structures. In contrast, Scap does not have to set such limits because it uses a dynamic memory man-



(a) Lost streams

(b) CPU utilization

(c) Software interrupt load

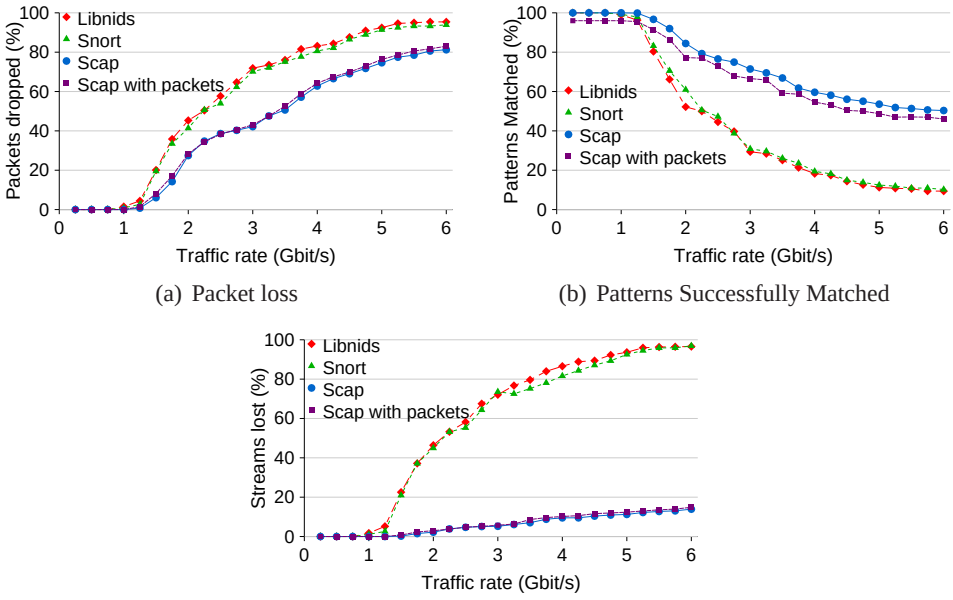
FIGURE 7.5: Performance comparison of Snort, Libnids, and Scap, for a varying number of concurrent streams.

agement approach: when more memory is needed for storing `stream_t` records, Scap allocates dynamically the necessary memory pools to capture all streams. In case an attacker tries to overwhelm the Scap flow table, Scap will use all the available memory for `stream_t` records. When there is no more free memory, Scap's policy is to always store newer streams by removing from the flow table the older ones, i.e., streams with the highest inactivity time based on the access list.

7.5.5 Pattern Matching

In the following experiments, we measure the performance of Scap with an application that receives all streams and searches for a set of patterns. We do not apply any cutoff so that all traffic is delivered to the application, and a single worker thread is used. Pattern matching is performed using the Aho-Corasik string matching algorithm [9]. We extracted 2,120 strings based on the `content` field of the "web attack" rules from the official VRT Snort rule set [7], and use them as our patterns. These strings resulted in 223,514 matches in our trace.

In this experiment We compare Scap with Snort and Libnids using the same string matching algorithm and set of patterns in all three cases. To ensure a fair comparison, Snort is configured only with the Stream5 preprocessor enabled, which performs transport-layer stream reassembly, using a separate Snort rule for each of the 2,120 patterns, applied to all traffic, so that all tools end up using the same automaton. The Scap and Libnids programs load the 2,120 patterns from a file,



(a) Packet loss

(b) Patterns Successfully Matched

(c) Lost streams

FIGURE 7.6: Performance comparison of pattern matching for Snort, Libnids, and Scap, for varying traffic rates.

build the respective DFA, and start receiving streams. We use the same chunk size of 16KB for all tools.

Figure 7.6(a) shows the percentage of dropped packets for each application as a function of the traffic rate. We see that Snort and Libnids process traffic rates of up to 750 Mbit/s without dropping any packets, while Scap processes up to 1 Gbit/s traffic with no packet loss with one worker thread. The main reasons for the improved performance of Scap are the improved cache locality when grouping multiple packets into their respective transport-layer streams, and the reduced memory copies during stream reassembly.

Moreover, Scap drops significantly fewer packets than Snort and Libnids, e.g., at 6 Gbit/s it processes three times more traffic. This behavior has a positive effect on the number of matches. As shown in Figure 7.6(b), under the high load of 6 Gbit/s, Snort and Libnids match less than 10% of the patterns, while Scap matches five times as many: 50.34%. Although the percentage of missed matches for Snort and Libnids is proportional to the percentage of dropped packets, the accuracy of the Scap application is affected less from high packet loss rates. This is because Scap under overload tends to retain more packets towards the beginning of each stream. As we use patterns from web attack signatures, they are usually found within the first few bytes of HTTP requests or responses. Also, Scap tries to deliver contiguous chunks, which improves the detection abilities compared to delivery of chunks with random holes.

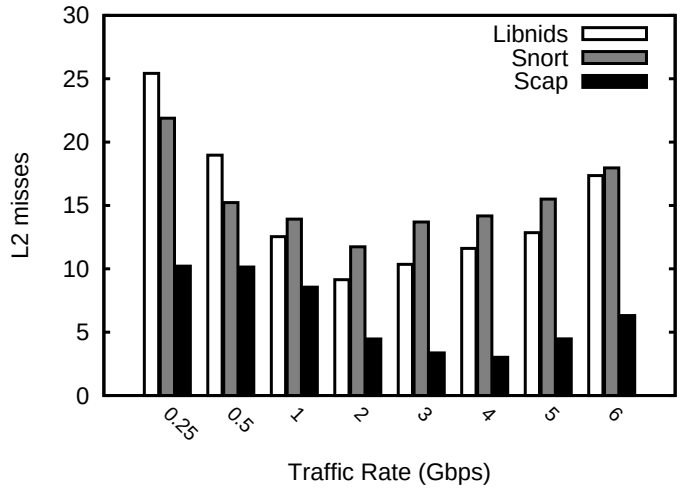


FIGURE 7.7: L2 cache misses of pattern matching using Snort, Libnids, and Scap, for varying traffic rates.

Favoring Recent and Short Streams

We turn our attention now to see how dropped packets affect the different stream reassembly approaches followed by Scap, Libnids, and Snort. While Libnids and Snort drop packets randomly under overload, Scap is able to (i) assign more memory to new or small streams, (ii) cut the long tails of large streams, and (iii) deliver more streams intact when the available memory is limited. Moreover, the Scap kernel module always receives and processes all important protocol packets during the TCP handshake. These packets may result in the creation of new streams, but they do not carry data to be stored. In contrast, when a packet capture library drops these packets due to overload, the user-level stream reassembly library will not be able to reassemble the respective streams, resulting in completely lost streams. Indeed, Figure 7.6(c) shows that the percentage of lost streams in Snort and Libnids is proportional to the packet loss rate (shown in Figure 7.6(a)). In contrast, Scap loses significantly less streams than the corresponding packet loss ratio. Even for 81.2% packet loss at 6 Gbit/s, only 14% of the total streams are completely lost.

Locality

Let's now turn our attention to see how different choices made by different tools impact locality of reference and, in the end, determine application performance. For the same pattern matching experiment, we also measure the number of L2 cache misses as a function of the traffic rate (Figure 7.7), using the processor's performance counters [6].

We see that when the input traffic is about 0.25 Gbit/s, Snort experiences about 25 misses per packet, Libnids about 21, while Scap experiences *half* of them: just

10.2 misses per packet. We have to underline that at this low traffic rate none of the three tools misses any packets, and we know that none of the tools is stressed, so they all operate in their comfort zone. The reason that Lidnids and Snort have twice as many cache misses as Scap can be traced to the better locality of reference of the Scap approach. By reassembling packets into streams from the moment they arrive, packets are not copied around: consecutive segments arrive together, are stored together, and are consumed together. On the contrary, Libnids and Snort perform packet reassembly too late: the segments have been stored in (practically) random locations all over the main memory.

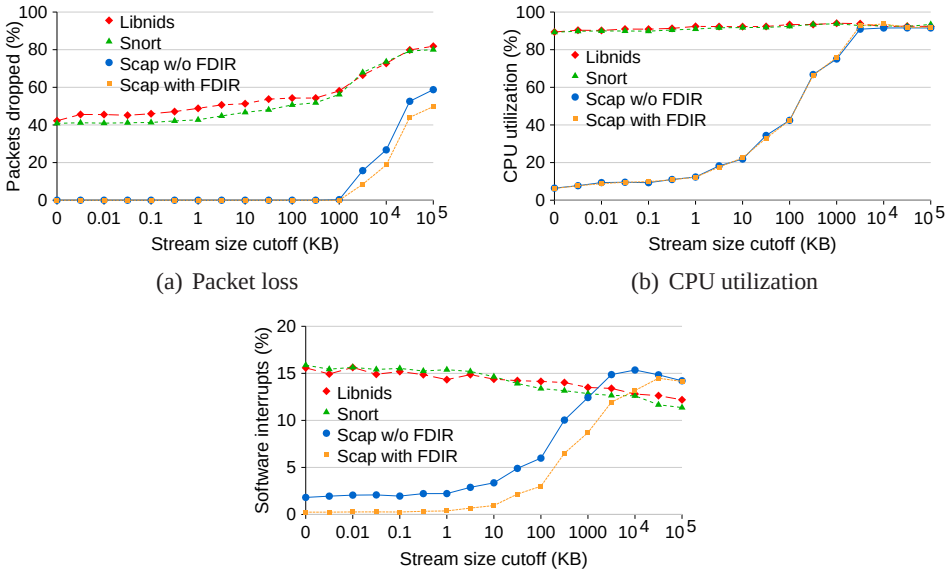
Packet Delivery

To evaluate the packet delivery performance of Scap, we ran the same application when Scap was configured with packet support, and pattern matching was performed on the delivered packet payloads. The results are shown in Figure 7.6 as well. We see that the performance of Scap remains the same when the pattern matching application operates on each packet, i.e., the percentages of dropped packets and lost streams do not change. We just observe a slight decrease in the number of successful matches, which is due to missed matches for patterns spanning the payloads of multiple successive packets.

7.5.6 Cutoff Points: Discarding Less Interesting Packets Before It Is Too Late

Several network monitoring applications need to receive only the initial part of each data flow [24, 86, 107], usually because they do not have the computing capacity to process the entire stream [107, 109]. Other systems, such as Time Machine [89], elevate the ability to store only the beginning of each flow into one of their fundamental properties. In this experiment, we set out to explore the effectiveness of Libnids, Snort, and Scap when implementing cutoff points. For Snort, we modified Stream5 to discard packets from streams that exceed a given cutoff value. Similarly, when the size of a stream reaches the cutoff value, Libnids stops the collection of data for this stream. In Scap, we just call the `scap_set_cutoff()` function in program's preamble using the desirable cutoff. We also compare Scap with and without using FDIR filters, which are added dynamically to the NIC for each stream when its cutoff is reached, to discard the rest of its packets at the card. The applications search for the same set of patterns as in the previous experiment.

Figures 7.8(a), 7.8(b), and 7.8(c) show the percentage of packet loss, CPU utilization, and software interrupt load as a function of the cutoff for a fixed traffic rate of 4 Gbit/s. Interestingly, even for a zero cutoff size, i.e., when all data of each flow is discarded, both Snort and Libnids experience as much as 40% packet loss, as shown in the left part of Figure 7.8(a). This is because Snort and Libnids first bring *all* packets to user space, and then discard the bytes they do not need. Indeed,



(a) Packet loss

(b) CPU utilization

(c) Software interrupt load

FIGURE 7.8: Performance comparison of Snort, Libnids, and Scap, for varying stream size cutoff values at 4 Gbit/s rate.

Figures 7.8(b) and 7.8(c) show that the total CPU utilization of Libnids and Snort is always close to 100% at this traffic rate irrespectively of the cutoff point.

In contrast, for cutoff points smaller than 1MB, Scap has no packet loss and very small CPU utilization. For instance, when Scap uses a 10KB cutoff, the CPU load is reduced from 97% to just 21.9%, as 97.6% of the total traffic is efficiently discarded. At the same time, 83.6% of the matches are still found, and no stream is lost. This outcome demonstrates how the stream cutoff, when implemented efficiently, can improve performance by cutting the long tails of large flows, and allows applications to keep monitoring the first bytes of each stream at high speeds. When the cutoff point increases beyond 1MB, CPU utilization reaches saturation and even Scap starts dropping packets. Enhancing Scap with hardware filters to discard packets within the NIC reduces the software interrupt load, and thus reduces the packet loss for cutoff values larger than 1MB.

7.5.7 Stream Priorities: Less Interesting Packets Are The First Ones To Go

To experimentally evaluate the effectiveness of Prioritized Packet Loss (PPL), we ran the same pattern matching application using a single worker thread while setting two priority classes. As an example, we set a higher priority to all streams with source or destination port 80, which correspond to 8.4% of the total packets in our trace. The rest of the streams have the same (low) priority. Figure 7.9 shows the

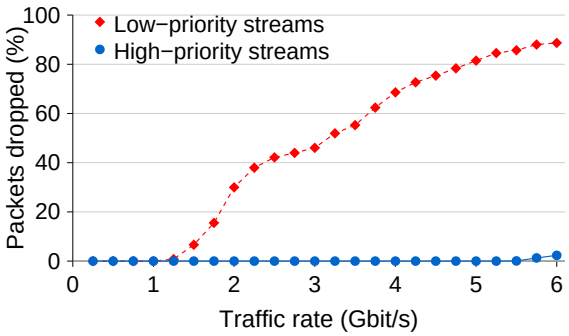


FIGURE 7.9: Packet loss for high- and low-priority streams, for varying traffic rates.

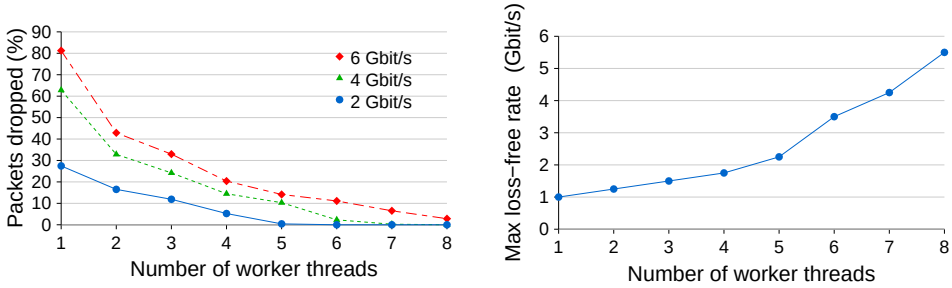
percentage of dropped packets for high-priority and low-priority streams as a function of the traffic rate. When the traffic rate exceeds 1 Gbit/s, the single-threaded pattern matching application cannot process all incoming traffic, resulting in a fraction of dropped packets that increases with higher traffic rates. However, we see that no high-priority packet is dropped for traffic rates up to 5.5 Gbit/s, while a significant number of low-priority packets are dropped at these rates—up to 85.7% at 5.5 Gbit/s. At the traffic rate of 6 Gbit/s, we see a small packet loss of 2.3% for high-priority packets out of the total 81.5% of dropped packets.

7.5.8 Using Multiple CPU Cores

In all previous experiments the Scap application ran on a single thread, to allow for a fair comparison with Snort and Libnids, which are single-threaded. However, Scap is naturally parallel and can easily use a larger number of cores. In this experiment, we explore how Scap scales with the number of cores. We use the same pattern matching application as previously, without any cutoff, and configure it to use from one up to eight worker threads. Our system has eight cores, and each worker thread is pinned to one core.

Figure 7.10(a) shows the packet loss rate as a function of the number of worker threads, for three different traffic rates. When using a single thread, Scap processes about 1 Gbit/s of traffic without packet loss. When using seven threads, Scap processes all traffic at 4 Gbit/s with no packet loss. Figure 7.10(b) shows the maximum loss-free rate achieved by the application as a function of the number of threads. We see that performance improves linearly with the number of threads, starting at about 1 Gbit/s for one worker thread and going all the way to 5.5 Gbit/s for eight threads.

The reason that we do not see a speedup of eight when using eight worker threads is the following: even though we restrict the user application to run on a limited number of cores, equal to the number of worker threads, the operating system kernel runs always on all the available cores of the processor. Therefore, when



(a) Packet loss

(b) Speedup

FIGURE 7.10: Performance of an Scap pattern matching application for a varying number of worker threads.

Scap creates less than eight worker threads, it is only the user-level application that runs on these cores. The underlying operating system and Scap kernel module runs on all cores.

7.6 Analysis

In this section, we analyze the performance of Prioritized Packet Loss (PPL) under heavy load, aiming to explore at what point PPL should start dropping low-priority packets so that high priority ones do not have to be dropped. For simplicity lets assume that we have two priorities: low and $high$. We define N to be $(memory\_size - base\_threshold)/2$. If the used memory exceeds N, then PPL will start dropping low priority packets. Given that N is finite, we would like to explore what is the probability that N will fill up and we will have to drop high-priority packets. To calculate this probability we need to make a few more assumptions. Assume that high-priority packet arrivals follow a Poisson distribution with a rate of λ, and that queued packets are consumed by the user level application. We assume that the service times for packets follow an exponential distribution with parameter μ. Then, the whole system can be modeled as an $M/M/1/N$ queue. The probability that all the memory will fill up is:

$$P_{full} = \frac{1-\rho}{1-\rho^{N+1}}\rho^N \tag{7.1}$$

where $\rho = \lambda/\mu$. Due to the PASTA property of the Poisson processes, this is exactly the probability of packet loss: $P_{loss} = P_{full}$.

Figure 7.11 plots the packet loss probability for high-priority packets as a function of N. We see that a memory size of a few tens of packet slots are enough to reduce the probability that a high-priority packet is lost to 10^{-8}. We note, however, that the speed with which the probability is reduced depends on ρ: the fraction of the high-priority packets over all traffic which can be served by the full capacity of the system. We see that when ρ is 0.1, that is, when only 10% of the packets are

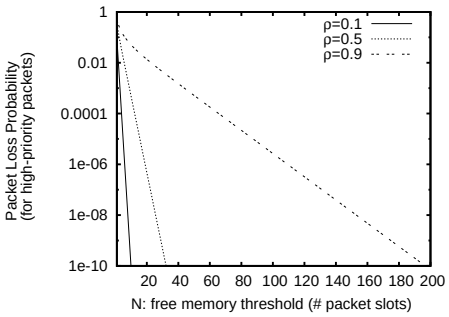


FIGURE 7.11: Packet loss probability for high-priority packets as a function of N.

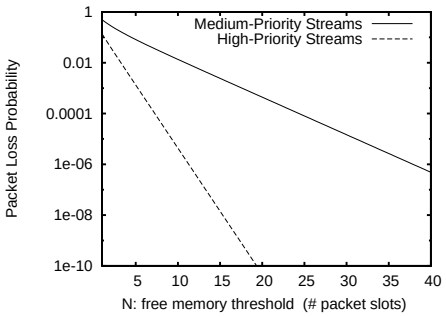
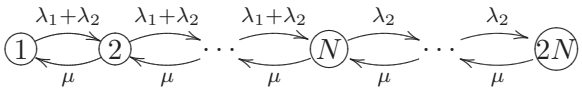


FIGURE 7.12: Packet loss probability for high-priority and medium-priority packets as a function of N.

high-priority ones, then less than 10 slots are more than enough to guarantee that there will be practically no packet loss. When ρ is 0.5 (i.e., 50% of the traffic is high-priority), then a little more than 20 packet slots are enough, while when ρ is 0.9, then about 150 packet slots are enough.

The analysis can be extended to more priority levels as well. Assume, for example, that we have three priority levels: low, $medium$, and $high$, that $N = (memory\_size - base\_threshold)/3$, that medium-priority packet arrivals follow a Poisson distribution with a rate of λ_1, and that high-priority packet arrivals follow a Poisson distribution with a rate of λ_2. As previously, assume that the service times for packets follow an exponential distribution with parameter μ. Then, the system can be described as a Markov chain with $2N$ nodes:



The packet loss probability for high-priority packets is:

$$P_{loss} = \rho_1^N \rho_2^N p_0 \tag{7.2}$$

where $\rho_1 = (\lambda_1 + \lambda_2)/\mu$, $\rho_2 = \lambda_2/\mu$, and

$$p_0 = 1/(\frac{1-\rho_1^{N+1}}{1-\rho_1} + \rho_1^{N/3}\frac{1-\rho_2^{N+1}}{1-\rho_2})$$

The packet loss probability for medium-priority packets remains:

$$P_{loss} = \frac{1-\rho_1}{1-{\rho_1}^{N+1}}{\rho_1}^N \tag{7.3}$$

Figure 7.12 plots the packet loss probability for high-priority and medium-priority packets as a function of N. We assume that $\rho_1 = \rho_2 = 0.3$. We see that a few tens of packet slots are enough to reduce the packet loss probability

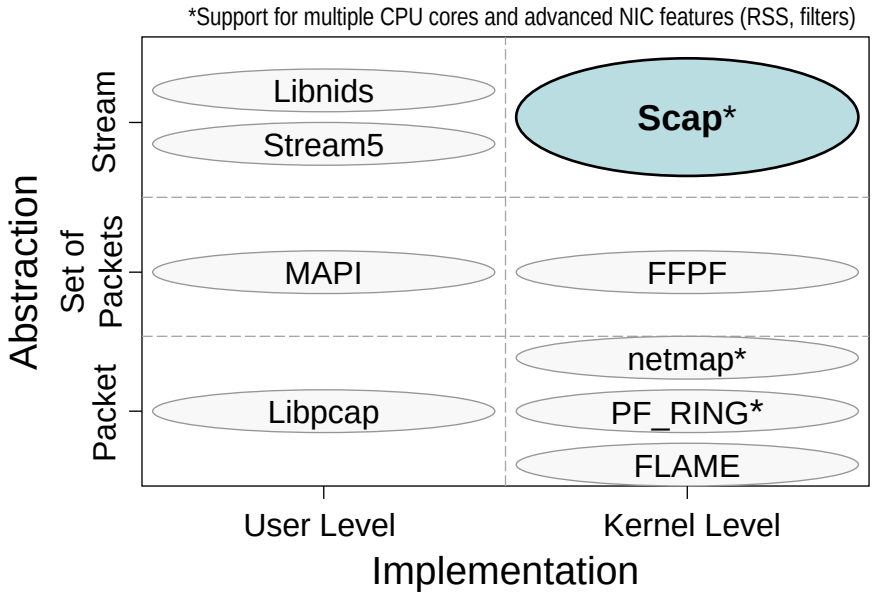


FIGURE 7.13: Categorization of network monitoring tools and systems that support commodity NICs.

for both high-priority and medium-priority packets to practically zero. Thus, we believe that PPL provides an effective mechanism for preventing uncontrolled loss of important packets in network monitoring systems.

7.7 Comparison With Other Capture Frameworks

There exists lots of related work in the area of traffic capture and analysis. To place our work in context, Figure 7.13 categorizes Scap and related works along two dimensions: the main abstraction provided to applications, i.e., packet, set of packets, or stream, and the level at which this abstraction is implemented, i.e., user or kernel level. Traditional systems such as Libpcap [92] use the *packet* as basic abstraction and are implemented in user level (bottom left of the figure). More sophisticated systems such as netmap [123], FLAME [10], and PF_RING [37] also use the packet as basic abstraction, but are implemented with kernel modifications to deliver better performance (bottom right of the figure). MAPI [143] and FFPF [19] use higher level abstractions such as the *set of packets*. Libnids and Stream5 provide the transport-layer *Stream* as their basic abstraction, but operate at user level and thus achieve poor performance and miss several opportunities of efficiently implementing this abstraction (top left of the figure). We see Scap as the only system that provides a high-level abstraction, and at the same time implements it at the appropriate level, enabling a wide range of performance optimizations and features.

7.8 Summary

In this chapter, we have identified a gap in network traffic monitoring: applications usually need to express their monitoring requirements at a high level, using notions from the transport layer or even higher, while most monitoring tools still operate at the network layer. To bridge this gap, we have presented the design, implementation, and evaluation of Scap, a network monitoring framework that offers an expressive API and significant performance improvements for applications that process traffic at the transport layer and beyond. Scap gives the stream abstraction a first-class status, and provides an OS subsystem for capturing transport-layer streams while minimizing data copy operations by optimally placing network data into stream-specific memory regions. It also offers a variety of features and performance optimizations, by (i) discarding uninteresting traffic efficiently within the kernel, (ii) reacting to overload conditions by dropping low priority traffic, (iii) utilizing multi-core architectures for parallel stream processing, and (iv) improving the memory locality and cache usage by grouping packets into steams. Using the Scap API, the user-level applications are able to communicate their stream-oriented needs directly to the underlying Scap kernel module.

The results of our experimental evaluation demonstrate that Scap is able to deliver all streams for rates up to 5.5 Gbit/s using a single core, two times higher than the other existing approaches. An Scap-based application for pattern matching handles 33% higher traffic rates and processes three times more traffic at 6 Gbit/s than Snort and Libnids. Moreover, we observe that user-level implementations of per-flow cutoff just reduce the packet loss rate, while Scap's kernel-level implementation and subzero copy eliminate completely packet loss for stream cutoff values of up to 1MB when performing pattern matching operations at 4 Gbit/s. This outcome demonstrates that cutting the long tails of large flows can be extremely beneficial when traffic is discarded at early stages, i.e., within the kernel or even better at the NIC, in order to spend the minimum possible number of CPU cycles for uninteresting packets. When eight cores are used for parallel stream processing, Scap is able to process 5.5 times higher traffic rates with no packet loss.

As networks are getting increasingly faster and network monitoring applications are getting more sophisticated, we believe that approaches like Scap, which enable aggressive optimizations at kernel-level or even at the NIC level, will become increasingly more important in the future.

8
Other Applications

In this chapter we explore how we can use similar approaches to solve two other problems related to network monitoring systems. First, we study the problem of energy efficiency in network monitoring systems, using NIDS as a case study. While building an energy-efficient NIDS, we identify an energy-latency tradeoff: while reducing the NIDS power consumption, the detection latency is significantly increased. We also explain how the increased detection latency impedes the timely reaction of a NIDS to the incoming attacks. To reduce the detection latency and resolve this tradeoff, we identify the most important packets for fast detection and we process them with higher priority.

In the second part of this chapter, we address the problem of long-term traffic recording using fixed-size storage. We propose the idea of *traffic aging*, to keep more traffic for recent time intervals and sample less traffic as it gets older. To select representative samples of the stored packets, we explore different sampling strategies such as random packet sampling, random flow sampling, and per-flow cutoff, to store less packets from the beginning of each flow.

8.1 Low-Power and Low-Latency Network Monitoring

Low power consumption is one of the main design goals in today's computer systems. Recently, much effort has been put into improving the energy efficiency in a variety of areas like data centers [132], high performance computing [36], mobile devices [112], and networks [63]. Towards this direction, we aim to build an energy-efficient Network-level Intrusion Detection System (NIDS). NIDS are commonly deployed to detect security violations, enhancing the secure operation of modern computer networks. They perform computationally heavy operations

like pattern matching, regular expression matching, and other types of complex analysis to detect at real time malicious activities in the monitored network. Thus, NIDS usually utilize multicore systems [114] or cluster of servers [84, 128, 146] to cope with increased link speeds and complicated analysis of network traffic.

Although NIDS are usually provisioned to operate at link rate, in order to be able to handle a fully utilized network (at the worst case), most networks are typically much less utilized than their maximum capacity. This results in increased power consumption at low traffic load. To reduce the energy spent under low traffic we aim at building a power-proportional NIDS using Dynamic Voltage and Frequency Scaling (DVFS) and sleep states (C-states), which can be found in modern processors. The system should consume the less power needed to sustain the incoming traffic load. We found that a NIDS consumes less power when it uses the smallest number of cores that can operate at the lowest possible frequency to process the network traffic, by keeping these cores nearly fully utilized. Our results indicate that this energy-efficient NIDS can process all packets with up to 23% lower power consumption than the original system at low rates. However, we observe a significant increase on the detection latency due to higher packet processing times when reducing the frequency, and mostly due to increased queuing delays imposed by the high utilization.

A low detection latency is very important for a NIDS in order to ensure a timely reaction to the attack. Upon the detection of a packet that carries an attack, the NIDS can actively terminate the offending connection or install a new firewall rule. This reaction should be immediate, before the attack packets reach the victim's machine and the attack succeeds. Therefore, our results indicate a new tradeoff for NIDS: the *energy-latency tradeoff*. Our key idea to resolve this tradeoff is to identify the most important packets for attack detection and process them with higher priority, resulting in low latency and fast detection. The rest packets are processed with lower priority to achieve an overall low power consumption.

We explore two alternative approaches to reduce the latency of high-priority packets: *time sharing* and *space sharing*. In time sharing we use a typical priority queue scheduling in each core. In space sharing the high-priority packets follow a different path, using dedicated cores with much lower utilization to achieve low latency. To implement space sharing we use features of modern network interface cards (NIC) to move efficiently the processing of least-significant packets to cores with higher utilization, a technique we call as *flow migration*. We experimentally compare the two approaches and we find that space sharing has a better power-latency ratio. This is because time sharing cannot efficiently reduce the queuing delays during a high utilization.

Based on these approaches we propose LEoNIDS: a NIDS architecture that resolves the energy-latency tradeoff. The implementation of LEoNIDS uses NIC features, a specialized kernel module, a modified user-level library, and it is based on the popular Snort NIDS [124]. LEoNIDS consumes less power, proportionally to the traffic load, while its detection latency remains low and almost constant at any traffic load.

8.1.1 Towards a Power Proportional NIDS

We first explore the design space to build a power-proportional NIDS.

Experimental Environment

Our testbed consists of two machines interconnected with a 10 GbE switch. Both machines are equipped with two six-core Intel Xeon E5-2620 processors with 15 MB L2 cache, 8 GB RAM, and an Intel 82599EB 10 GbE network interface. The clock frequency of these processors can be scaled from 1.2 GHz to 2.0 GHz using DVFS, which results in 9 available frequency steps (P-states). They also support Intel Turbo Boost technology to further increase their frequency up to 2.5 GHz. To reduce power consumption, each idle core can be put independently into one of the 3 available sleep states: C1, C3 or C6. We measure the power consumption in the NIDS machine using the Watts up? PRO ES device, by sampling and storing the power at one second intervals. All our measurements run for significantly higher time periods than one second.

The first machine is used for traffic generation. The generated traffic reaches the second machine, which runs Snort IDS [124] v2.8.3.2 with official rule set [7] containing 8308 rules. We use PF_RING [55] v5.3.0 and ixgbe driver v3.7.17 to split the incoming traffic to active cores using the Receive Side Scaling (RSS) [75] feature of Intel 82599 NIC [74]. We set the size of the ring buffer that stores packets at each core to 4096 slots. To change the frequency we use the *cpufrequtils* package. Both machines run 64-bit Linux (kernel version 3.5.0).

We generate real traffic by replaying an one-hour long anonymized trace captured at the access link of an educational network. The trace contains 58,714,906 packets and 1,493,032 flows, totaling more than 40GB, 95.4% of which is TCP traffic. For this trace Snort triggers 1851 alerts from 76 different rules. Most of the matching rules are related to common threats and protocol violations. In order to strengthen our evaluation, we augmented the trace with 120 traces of real attacks captured in the wild [117], adding 233 more alerts from 14 different rules.

Power Consumption

The system's idle power consumption is 85.1 W, and when Snort fully utilizes all cores it consumes 145.7 W. Thus, we estimate that the extra power from idle state is consumed by the NIDS. As NIDS perform heavy computational operations, the CPU consumes the larger portion of energy in the system. We measure the CPU power consumption by accessing the RAPL (Running Average Power Limit) registers provided by each Intel Xeon E5-2620 CPU, which measure the total energy consumed by each chip. Varying the traffic load results in different NIDS utilizations. In all NIDS utilizations, 58-62% of the total power is consumed by the two CPUs.

Modern processors offer two ways to reduce power consumption: frequency scaling (DVFS), and sleep states (C-states). Intel processors have a single volt-

Active cores	Frequency	Power consumption	Detection latency
6	2.0 GHz	107.0 W	0.371 ms
8	1.5 GHz	104.2 W	0.856 ms
10	1.2 GHz	100.2 W	1.228 ms

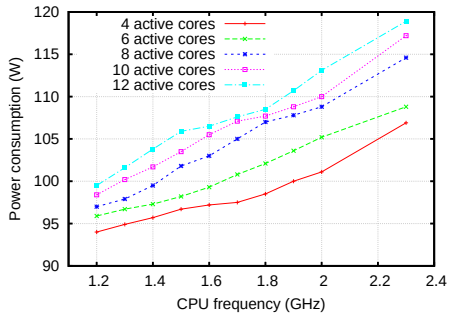
TABLE 8.1: Using more cores at lower frequency consumes less power but results in higher detection latency. (when processing 1.5 Gbit/sec).

age and frequency regulator, so the frequency changes uniformly at all cores of a processor. However, each core can operate in a different C-state to save energy. The power consumption of each core consists of (i) active power consumed when the core processes packets at the current frequency, (ii) power consumed to enter a C-state, and (iii) power consumed during the idle state. We see that idle cores consume less power when they are in C6 state, so we put inactive cores in this state. There is an increased latency to wake up cores from C6 state, so we need to activate them few microseconds before this core will be necessary.

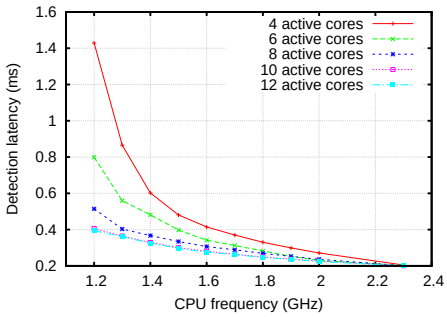
Based on the packet arrival rate, we aim to find the most energy-efficient strategy for a NIDS by properly adapting the frequency and the number of active cores (not in C-states). The two main questions are: (i) is it better to operate at lower frequency or utilize sleep states? (ii) is it better to use more cores on lower frequency or less cores at higher frequency?

To find the optimal strategy we measure Snort's power consumption as a function of the CPU frequency and the number of active cores, when sending traffic at a constant rate of 0.6 Gbit/sec. Figure 8.1(a) shows that the lowest power consumption is achieved when using 4 active cores at 1.2 GHz, which is the minimum setup that is able to handle the 0.6 Gbit/sec traffic with no packet loss. In this setup we see up to 21% reduced power consumption compared to 12 cores at the maximum frequency.

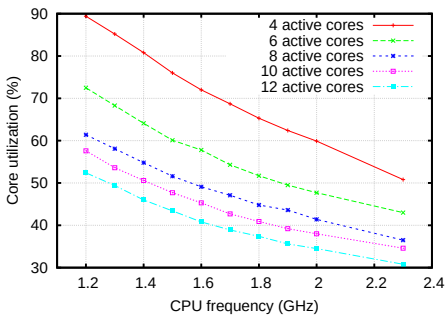
We observe that the less power is consumed when the system operates at the lowest possible frequency with no idle time, instead of running at higher frequencies and entering C-states during idle periods. Moreover, we see that using more cores at lower frequency is more energy efficient than using less cores at higher frequencies. For instance, Table 8.1 shows three alternative setups that can be used to process 1.5 Gbit/sec, as they offer approximately the same computing power: 6 cores at 2.0 GHz, 8 cores at 1.5 GHz, or 10 core at 1.2 GHz. We see that 10 cores at 1.2 GHz consume less power than the other cases. Figure 8.1(c) shows the average utilization of active cores. We see that power consumption decreases as the core utilization increases and approaches 100%. This is because being idle is not sufficiently efficient, i.e., the power consumed to enter and leave C-states and during these idle periods is quite significant.



(a) Power consumption



(b) Detection latency



(c) Utilization per active core

FIGURE 8.1: Fewer cores and lower frequency reduce the power consumption but increase the detection latency. Power consumption, detection latency, and core utilization as a function of frequency and number of active cores when running Snort and sending constant traffic at 0.6 Gbit/sec. We see that power consumption decreases as the utilization of active cores approaches 100%. However, this results in increased detection latency.

Adapt to the Traffic Load

Our results indicate that a power-proportional NIDS should utilize the smallest number of cores that are able to sustain the incoming traffic without any packet loss when they operate at the lowest possible frequency. Therefore, the system should dynamically adapt to the traffic load by changing the frequency and activating/deactivating cores. We observe that it is preferable to first activate cores of the same CPU, which explains the larger distance in the power consumption between 6 and 8 cores in Figure 8.1(a).

A NIDS is based on the underlying packet capturing system to receive packets for processing. To tolerate processing spikes or short-term overloads, the packet capturing system is able to store a limited number of packets in memory queues (ring buffers). Modern NICs [74] offer multiple receive queues and are able to distribute the packets among them to allow for efficient multicore processing [55]. Thus, a packet capturing system with multicore support uses a separate queue per

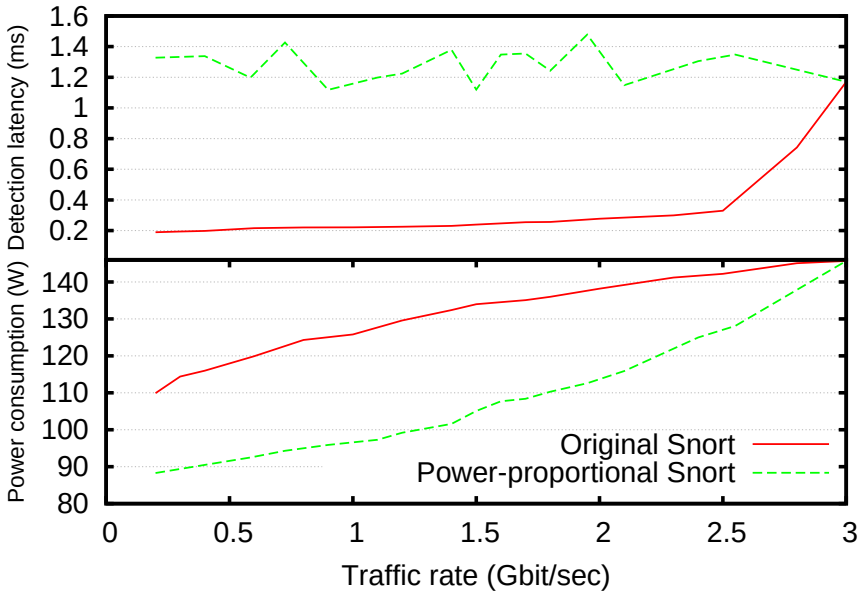


FIGURE 8.2: A straight-forward power-proportional NIDS consumes less power with higher detection latency. Power consumption and detection latency of a straight-forward power-proportional NIDS versus the original NIDS as a function of traffic rate.

core. When queues are getting full, the system has a strong indication of higher load than it can handle with the current setup, so it needs to employ more cores or increase the frequency. A straight-forward power-proportional NIDS uses the following strategy:

1. The system starts with a single active core at the minimum frequency.

2. It continuously monitors the queues' usage.

- **2.1.** If queues are filled by more than a *high threshold*:
  - **2.1.1.** If there are inactive cores, it wakes up one more core.
  - **2.1.2.** Else, it increases the frequency of all cores to the next step.
- **2.2.** If queues are filled by less than a *low threshold*:
  - **2.2.2.** If lowest frequency is used, it deactivates one core.
  - **2.2.2.** Else, it decreases the frequency to the previous step.

We implemented this online adaptation algorithm within the packet capturing subsystem, as a Linux kernel loadable module, and we ran Snort over this system while varying the load. We set *high threshold* to 90% and *low threshold* to 70%. Figure 8.2 (bottom part) shows the power consumption of this straight-forward energy-efficient NIDS as a function of the traffic rate, compared to the original

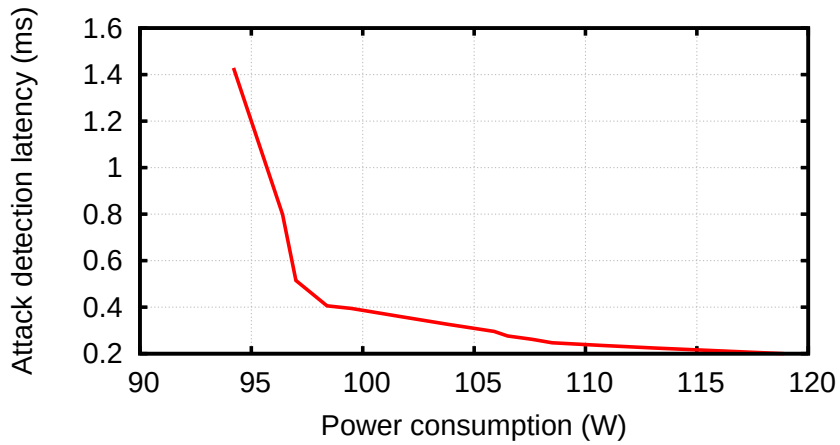


FIGURE 8.3: The energy-latency tradeoff. Detection latency as a function of power consumption when sending 0.6 Gbit/sec traffic. We see that detection latency increases significantly as power consumption is reduced.

system. We see that the vanilla system consumes 24% less power when processing 0.2 Gbit/sec, compared with the power consumption at 3 Gbit/sec. Contrary, the power-proportional NIDS adapts much better to the load reducing the power consumption by 39% when processing 0.2 Gbit/sec. In low rates, it consumes up to 23% less power than the original Snort.

8.1.2 The Energy-Latency Tradeoff in NIDS

Although a power-proportional NIDS is able to handle the same traffic as the original system with lower energy consumption, we would like to explore the impact of this approach on the detection latency.

Detection Latency

We instrumented Snort to measure the attack detection latency, by subtracting from the time that an alert is triggered the timestamp of the packet that contains the attack. The packet's timestamp is set within the packet capturing module before the packet is queued. Figure 8.1(b) shows the detection latency as a function of frequency and number of active cores for 0.6 Gbit/sec traffic. We see a linear increase when frequency is reduced up to 1.6 GHz and up to 8 cores are used, but we see an exponential increase to the detection latency when core utilization exceeds 70%. To better see the relation between power consumption and detection latency we replot these data in Figure 8.3. We see a clear tradeoff: to achieve power consumption lower than 100 W, the detection latency should be increased 2–7 times.

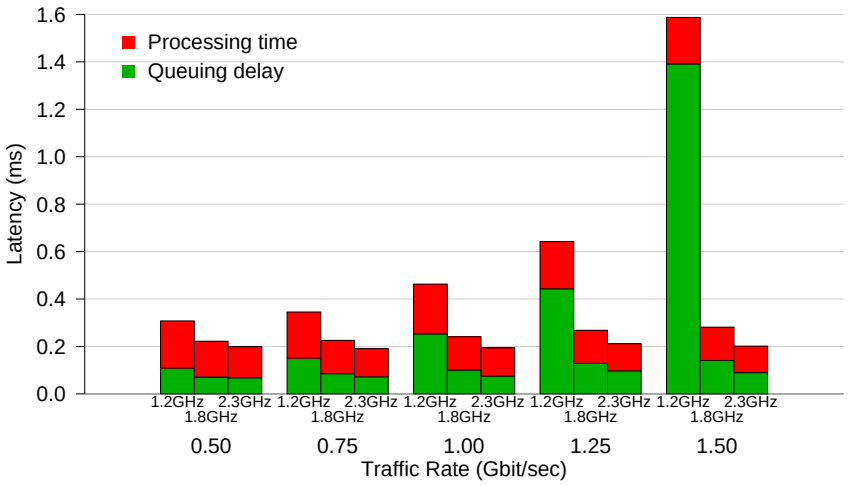


FIGURE 8.4: The main cause of increased detection latency is higher queuing delay. Processing time and queuing delay as a function of traffic rate. We see that for low frequency and high rates, when the latency significantly increases, queuing delay is much higher than processing time.

Table 8.1 leads us to the same outcome: although using 10 cores at 1.2 GHz consumes the less power, it comes at a price of significantly increased latency. Figure 8.2 (upper part) shows the detection latency of a power-proportional NIDS, compared to the original system. We see that although it consumes less power, it has a significantly higher detection latency at all rates. This is because this system selects the frequency and number of cores that lead to high utilization, close to 100%, in order to save energy. As a consequence, the detection latency remains always high.

Deconstructing Detection Latency

We define detection latency as the time passed from the arrival of the last packet that contains the attack till the alert generation in the NIDS. Thus, the detection latency is equal to the latency imposed per each attack packet, from the capturing time till it finished processing. The packet latency can be divided in three parts: (i) *interrupt handling time*, i.e., the time spent for packet handling in OS kernel, (ii) *queuing delay*, i.e., the time that packet waits in a queue to be delivered for processing, and (iii) *processing time* by the NIDS at user level. We see that the interrupt handling time per packet is negligible compared to queuing delay and NIDS processing time. Thus, the increased detection latency may occur due to higher processing times when reducing the frequency or due to higher queuing delays imposed by the increased utilization.

To explore why detection latency is increased, we measure how much each part contribute to the detection latency as we vary the offered traffic rate for different

frequencies. We instrumented Snort to measure (i) the queuing delay per packet, by subtracting the packet's timestamp from the time that packet is received in Snort for processing, and the packet's processing time in Snort. Figure 8.4 shows the average processing time and queuing delay per each attack packet for traffic rates ranging from 0.5 Gbit/sec to 1.5 Gbit/sec, when using 12 cores in 1.2 GHz, 1.8 GHz, and 2.3 GHz. In low frequency and high rates, when the system is more utilized, queuing delay is the main factor of the increased detection latency. For instance, when processing 1.5 Gbit/sec at 1.2 GHz, queuing delay is 7 times higher than processing time. This is because the higher utilization results in a large number of packets waiting at each queue and thus in an exponentially higher queuing delay. In contrast, processing time increases linearly as we decrease frequency.

8.1.3 Solving the Energy-Latency Tradeoff in NIDS

We aim to solve the energy-latency tradeoff using domain-specific knowledge on NIDS. This will enable us to build a NIDS with both low power consumption *and* low detection latency. The key idea of our approach is based on the fact that there is a small percentage of packets with significantly higher probability to contain an attack. These are the first few packets of each connection. Then we propose two alternative approaches to process these packets with low latency, while consuming less power proportional to the workload.

Identify the Most Important Packets for Detection Latency

One way to address the energy-latency tradeoff in NIDS would be to keep the utilization of active cores within a specific range, so to keep power consumption and detection latency lower than the respective thresholds. However, in this way a NIDS cannot achieve the lowest possible power consumption, while detection latency may also be much higher. To efficiently resolve the tradeoff, we use domain-specific knowledge about NIDS: we capitalize on the fact that not all packets have the same probability to carry an attack. By identifying the most interesting packets, a NIDS is able to process them with higher priority to achieve fast detection, while efficiently reducing the power consumption at each traffic load at the same time.

A key abstraction we use to identify the most important packets for attack detection is the network *flow*: a flow is defined as the set of packets belonging to the same one-way connection, i.e., packets with same protocol, source and destination IP addresses and port numbers (5-tuple). Previous works have shown that most attacks are found among the first few bytes of each flow [86, 89, 107]. This is because many types of threats like port scanning, service probes and OS fingerprinting, code-injection attacks, and brute force login attempts, require a new connection for each attempt, and the attack vector is found in the first few bytes of the flow. In contrast, very large streams usually correspond to file transfers, VoIP communication, or streaming media applications, which typically are not related to security threats. Due to the heavy-tailed flow size distribution in the Internel [53],

Attack category	Labeled attacks	Background traffic attacks
Web attacks	14	635
Specific threats	15	282
IMAP & SMTP	1	172
NetBIOS & RPC	200	236
Attack responses	0	213
SQL & MySQL	0	188
Other (spyware, backdoor, misc)	3	125

TABLE 8.2: Classification of the attacks detected in our trace.

the first bytes of each flow correspond to a very small percentage of the total traffic. Thus, processing the respective packets with higher priority and lower latency will result in faster detection for most attacks.

To validate and analyze our choice for high-priority packets in a NIDS, we measure the position of each attack within its flow, for attacks detected while running Snort with our trace. As we explained in Section 8.1.1, we injected 233 real attacks into the trace (labeled attacks), while the background traffic contains 1851 more attacks. Most of these attacks are related to popular threats and protocol violations. Table 8.2 presents a classification of these attacks based on Snort's ruleset [7].

Figure 8.5 shows the CDF of the detected attacks' position within their flows. We see that 50% of the attacks are found within the first 2 KB of a flow, while 90% of the attacks are detected in the first 30 KB of their flows. Only 2% of the attacks are found beyond the first 200 KB. We observed that the labeled attacks, which we consider more important as they correspond to real attacks and have been validated as true positives, are always detected within the first 5 KB of their flows. We found that the small percentage of attacks detected beyond 100 KB of a flow correspond to less significant threats and are usually triggered by threshold-based rules. Thus, the first few bytes of each flow have a much higher probability to actually contain an attack. We can separate the respective packets by applying a cutoff value to the flow size. Then we classify as high priority the packets until this cutoff.

Figure 8.5 also presents the CDF of the fraction of traffic that is located in a flow before the corresponding position on the x-axis. This fraction is the percentage of high-priority traffic as a function of the cutoff applied. For instance, 10% of the total traffic is found in the first 500 KB of the flows. This means that a cutoff value of 500 KB per flow will classify 10% of the total traffic as high-priority, and 99% of the attacks can be detected on this high-priority traffic.

Tolerating Evasion Attempts

An attacker could try to exploit the flow cutoff mechanism used for priority assignment in order to increase detection latency and impede a timely reaction. Thus, we aim to protect LEoNIDS against such attacks. One way to exploit the cutoff mechanism would be to overburden the system with high-priority packets, e.g., by

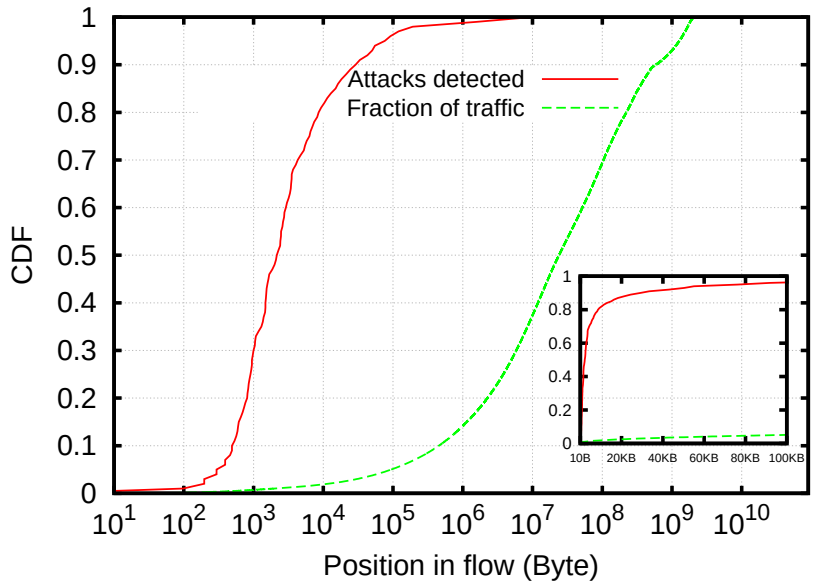


FIGURE 8.5: Most attacks are detected within the first few KBs of a flow, which is a small fraction of the total traffic. CDF of attack's position at each flow and fraction of traffic per each position.

sending a large number of small flows. However, as we explain in the following sections, LEoNIDS properly adapts to the traffic load by increasing frequency and active cores so that the latency of high-priority packets remains always low. In the worst case, e.g., a fully utilized system only with high-priority packets, LEoNIDS will approach the behavior of the original system: it may spend the maximum available power to keep latency of high-priority packets low.

Another way for an attacker to exploit our cutoff-based approach would be to push the attack into low-priority packets, resulting in higher detection latency. To address this attack, we take a number of countermeasures. We define flow cutoff in bytes, not in packets, so an attacker cannot exceed it by sending small packets. To handle persistent connections, like HTTP keep-alive connections, we reset flow size to zero for each new request or response. Finally, we use a lower limit for the flow cutoff value. This is because most protocol implementations have a maximum protocol message (request/response) and headers size, and may close connections exceeding the size. Thus, putting the attack beyond this size is not always possible.

For instance, many attacks are detected at the HTTP protocol, usually based on a signature matching in URI or request headers. Although attackers can send an arbitrary large URI to exceed cutoff, e.g., by adding KBs of space characters before URI, which are stripped by servers, or by adding dummy parameters with large values, all Web servers have a *maximum URI size* configuration option. Similarly, the have a *maximum request size* option). When a URI exceeds this limit, an *HTTP/1.1 414 Request-URI Too Large* error is returned, and request is not processed. Hence,

the actual attack cannot succeed. Similarly, when the *maximum request size* limit is exceeded, servers respond with `HTTP/1.1 400 Bad Request (Header Field Too Long)`.

In most Web servers, the default maximum URI size is 8 KB. Thus, using a cutoff larger than 8 KB ensures the timely detection of all *successful* attacks against servers using this limit. To find out how many of the popular Web servers use this default limit, we sent a request with URI slightly larger than 8 KB to the top-100 Web sites based on the ranking of *alexa.com*. The 98 of them responded with an `HTTP/1.1 414 Request-URI Too Large` error, while only two of them accepted the request. When sending requests with 100 KB long URI, all the top-100 Web sites responded with error. Similarly, other protocols (e.g., IMAP, SMTP, NetBIOS) have also a maximum message size. Even if it is equal to few MBs, the fraction of high-priority traffic remains low. Another reason for setting a lower limit for cutoff value is that 49% of the Snort rules in our ruleset use the *depth* keyword: these rules require a pattern to be detected in a specific distance from the beginning of a packet or flow.

Priority Enforcement

Since we have identified the most important packets for fast detection, we need to ensure a low latency for these packets when the system enters into a power saving mode and active cores' utilization increases. We propose two alternative techniques to ensure low latency for the high-priority packets: *time sharing* and *space sharing*.

Time Sharing

Time sharing uses a typical priority queue scheduling to favor the high-priority packets. It first classifies packets into flows and then uses a flow cutoff to assign them a low or high priority. Then, packets are stored into the respective priority queue. When a new packet is scheduled for processing, the NIDS choose the next packet from the high-priority queue. If this queue is empty, a low-priority packet is chosen. However, this priority queue scheduling is non preemptive: when a high-priority packet arrives and a low-priority packet is being processed, the NIDS cannot evict the low-priority packet to serve immediately the high-priority packet. Time sharing follows the same strategy described at section 8.1.1 to adapt frequency and number of cores.

Space Sharing

In time sharing, the cores of the energy-efficient NIDS remain almost fully utilized. This may cause reduced performance due to the non-preemptive priority queue scheduling. In space sharing, we use separate cores for each priority. We aim to keep cores that serve high-priority packets less utilized, to ensure low latency. In contrast, cores serving low-priority packets can remain highly utilized to allow for

reduced power consumption. The increased latency for low-priority packets is less likely to affect the overall detection latency. As the majority of the packets have low priority (see Figure 8.5), most cores can be used to serve low-priority packets with high utilization that is necessary to achieve significant energy savings.

In order to reduce even more the detection latency, we would like to increase the frequency of the dedicated cores used to serve high-priority packets. However, the single per chip regulator in out Intel processors limits significantly our ability to change the frequency of high priority cores independently of low priority cores. Fortunately, our analysis in section 8.1.2 shows that core utilization is the main factor of an increased detection latency. Thus, just reducing the utilization could be enough to achieve our low latency goal even with a lower frequency, which is necessary for low priority cores to reduce their power consumption.

Space sharing is based on two main ideas: *flow migration* and *adaptive core management*.

Flow migration. The flow migration technique, assisted by advanced features of modern NICs, is used to distribute efficiently the packets into cores based on their priority. Initially, all packets arrive at the high-priority cores. Then, packets are classified into flows. When a flow size exceeds the specified cutoff value, the flow is moved into a low-priority core by instructing the NIC to schedule all the successive packets of this flow into this core. Thus, only the high-priority packets remain for processing into the high-priority cores. The low-priority packets are moved to the rest cores using the flow migration technique.

Adaptive core management. Space sharing dynamically partitions the active cores into high-priority and low-priority cores, based on the workload. It uses the optimum number of high-priority cores that keep their utilization within a desirable range. Using more cores than necessary may increase power consumption, while less cores may increase detection latency. Therefore, we propose the following adaptive core management algorithm, which extends the core/frequency adaptive algorithm we presented in Section 8.1.1:

1. The system starts with one high-priority and one low-priority core.

2. It continuously monitors the queues' usage.

- **2.1.** If high-priority queues are filled by more than a *high-priority up threshold*:
  - **2.1.1.** If exist inactive cores, activate a high-priority core.
  - **2.1.2.** Else increase the frequency.
  - **2.1.3.** If maximum frequency is used, reduce flow cutoff until it reaches a certain limit.
- **2.2.** If high-priority queues are filled by less than a *high-priority down threshold*:
  - **2.2.3.** Increase cutoff up to a certain limit.

2.2.1. Else reduce the frequency.

2.2.2. If lowest frequency is used, deactivate a high-priority core.

2.3. If low-priority queues are filled by more than a *low-priority up threshold*:

2.3.1. If exist inactive cores, activate a low-priority core.

2.3.2. If all cores are used, increase the frequency.

2.4. If low-priority queues are filled by less than a *low-priority down threshold*:

2.4.1. Reduce the frequency.

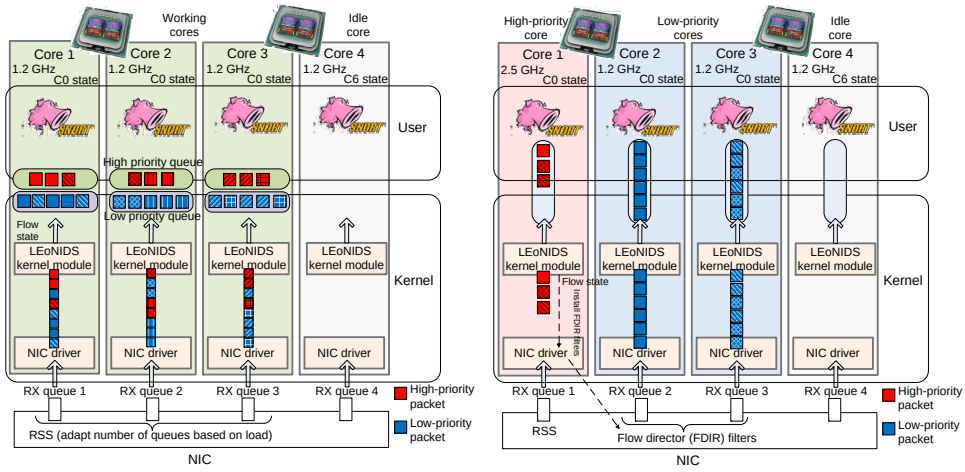
2.4.2. If lowest frequency is used, deactivate a low-priority core.

The *high-priority up threshold* ensures a low utilization for high-priority packets. The *low-priority up threshold* ensures that no packet will be lost. We can also control the load of high- and low- priority cores by changing the cutoff value, which divides the traffic into high- and low-priority packets. However, decreasing the flow cutoff is not always a good choice, as the probability that an attack occurs in a low-priority packet increases. Thus, we keep the cutoff always within a certain range.

8.1.4 Implementation

Based on the two alternative approaches we implemented LEoNIDS: a NIDS architecture that offers both low power consumption, proportionally to the load, and low detection latency. Figure 8.6 illustrates the architecture of LEoNIDS with time sharing and space sharing. Our implementation utilizes advanced features of modern NICs, and it is based on a specialized kernel module that modifies the packet capturing subsystem. Moreover, it includes a modified user-level packet capturing library and slight modifications to Snort NIDS [124].

We implemented the online frequency adaptation and core management algorithm within the packet capturing subsystem as a Linux kernel loadable module. Both time sharing and space sharing are implemented within this module. The module runs as a protocol handler and processes all captured packets. It monitors the packet queues per each core and properly adapts the number of active cores and the CPU frequency. This module is also responsible to store packets in the proper queues and impose a scheduling or load balancing policy. The packets are distributed among the available cores either with the RSS hash-based load balancing scheme [75] or with a dynamic load balancing scheme using the flow director filters (FDIR), which are used to define the core that will serve each flow. We deliver packets at user-level through memory mapped buffers, and we built a libpcap [92] wrapper library. Then, we link Snort with this user-level library, instead of the original libpcap.



(a) LEoNIDS with time sharing (b) LEoNIDS with space sharing

FIGURE 8.6: The LEoNIDS architecture with time sharing and space sharing.

Time Sharing

In time sharing we extend the ring buffers of the packet capturing system using a typical priority queue scheme. The incoming packets are classified into flows and are assigned a low or high priority. Based on its priority, each packet is stored in the proper queue. The modified user level library reads the next packet from the high priority queue, and only if it is empty, from the low priority queue. This packet is then delivered to Snort for processing.

Space Sharing

In space sharing we use dedicated cores to process the high-priority packets with reduced latency. We aim to keep the utilization of these cores between 30%–50%, which results in low queuing delays as we see in Sections 8.1.2 and **??**. Based on the queue utilization we properly adapt flow cutoff, number of high-priority cores, and frequency. The RSS uses a redirection table to distribute the incoming packets to the available cores. To implement space sharing, we first modify the redirection table so that RSS splits all packets only to high-priority cores. Then, these cores classify packets into flows. When a flow exceeds the cutoff size, an FDIR filter is added to the NIC in order to move the processing of this flow to a low-priority core (flow migration). The low-priority core is chosen in a round-robin fashion. Each flow that exceeds the cutoff value moves from one core to another only once, so the cache locality is not significantly affected. Using the FDIR filters for flow migration is highly efficient and improves cache performance, as each core accesses only its local data. We keep a list with all filters that are installed at the NIC, so when a flow expires (either explicitly by a TCP RST/FIN packet, or by an inactivity timeout) the respective FDIR filter is removed by the

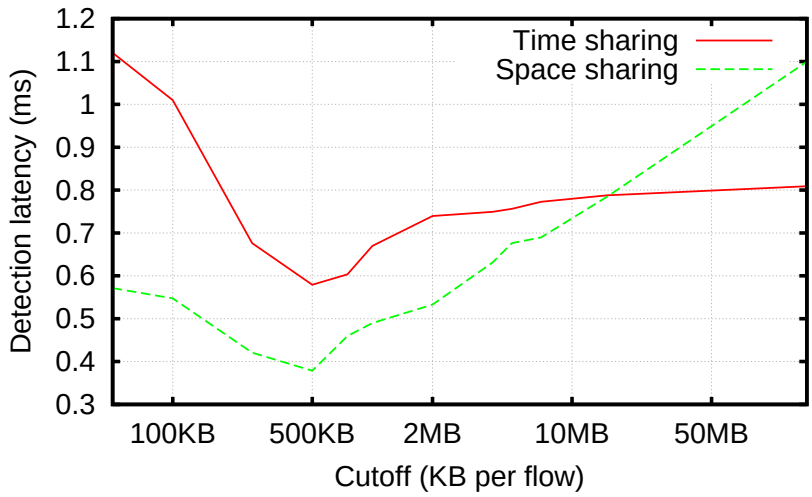


FIGURE 8.7: The optimal cutoff for time sharing and space sharing. Detection latency as a function of cutoff. In both time sharing and space sharing we see the lowest detection latency for 500 KB per flow.

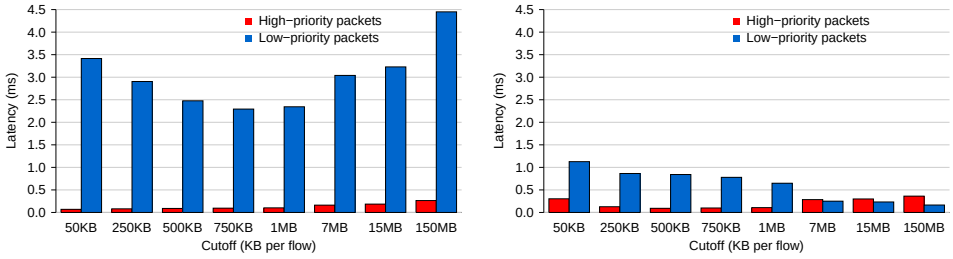
NIC. The intel 82599 NIC [74] offers up to 8K perfect match and 32K signature-based FDIR filters. In case all filters are used, space sharing evicts the oldest filter to accommodate a new flow.

8.1.5 Experimental Evaluation

Comparing Time Sharing with Space Sharing

Finding the optimal cutoff. Using a small cutoff reduces the percentage of high-priority packets, and thus their queue utilization and queuing delays. However, the probability that an attack will be found in low-priority packets, which experience a higher delay, increases. To find out the optimal cutoff for time sharing and space sharing, we vary the cutoff values from 50 KB to 150 MB per flow while sending constant traffic at 1.0 Gbit/sec. Figure 8.7 shows that the optimal cutoff for both approaches is close to 500 KB. Using this cutoff, 99% of the attacks reside into the high-priority packets. For lower cutoff values, more attacks are found in low-priority packets with increased detection latency, while higher cutoff values increase the queuing delay of high-priority packets. We also see that space sharing achieves a lower detection latency for all cutoff values below 20 MB, up to 50% lower (for 50 KB cutoff) and 35% lower for the optimal cutoff of 500 KB per flow.

The effect of priority on latency. To better understand the detection latency we observed, we explore how the packet's latency (queuing plus processing time) changes for each priority with different cutoff values. Figure 8.8 shows the latency of high- and low-priority packets for time and space sharing as a function of cutoff when sending at 1.0 Gbit/sec. In time sharing, we see that low-priority packets



(a) Time sharing

(b) Space sharing

FIGURE 8.8: Low-priority packets in time sharing experience a much higher latency that high-priority packets. Latency of high- and low-priority packets for time sharing and space sharing as a function of cutoff when sending at 1.0 Gbit/sec. We see that both high-priority and (especially) low-priority packets experience lower latency in space sharing comparing with time sharing.

experience up to 49.3 times higher latency than high-priority packets. As cutoff increases, we see a slight increase on the latency of high-priority packets due to the larger number of packets arriving at high-priority queues. Contrary, the latency of low-priority packets significantly decreases until cutoff reaches 750 KB, because the fraction of low-priority packets decreases, resulting in much less utilization in low-priority queues. When cutoff increases above 750 KB, the latency of low-priority packets increases fast. This is because they the low-priority packets wait for an increasing number of high-priority packets to be processed.

In space sharing, we see a much lower difference between the latency of low- and high-priority packets. Note that both low- and high-priority packets experience lower latency compared to time sharing. Especially the latency of low-priority packets is significantly lower and clearly decreases as cutoff increases. This is because low- and high-priority packets are processed in parallel in different cores, and the fraction of low-priority packets decrease with higher cutoff values. The latency of high-priority packets is also reduced, as space sharing is able to keep high-priority cores less utilized. As the fraction of high-priority packets increase with cutoff, we see a slight increase on their latency for higher cutoff values. The increased latency in high-priority packets for very small cutoff values is due to the overhead of the very often FDIR establishments.

Comparing All Approaches

Varying the load. We now compare all approaches, i.e., (i) the original Snort, (ii) the straight-forward power-proportional NIDS we described in section 8.1.1, (iii) LEoNIDS with time sharing, and (iv) LEoNIDS with space sharing, in terms of both detection latency and energy efficiency when varying the traffic load. In time sharing we use a 500 KB cutoff, which was found to perform better. In space sharing we use an adaptive cutoff that ranges from 300 KB to 1 MB, i.e., close to

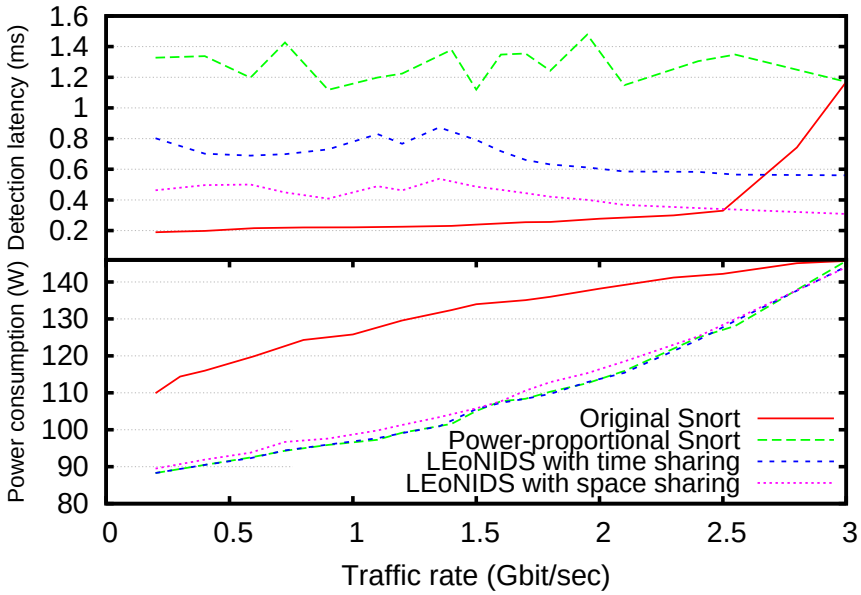


FIGURE 8.9: Space sharing offers the best power-latency ratio. Power consumption and detection latency of all approaches as a function of traffic rate. We see that LEoNIDS with space sharing consumes the same power with other power-proportional approaches, but with significantly lower detection latency. Compared to the original system, space sharing consumes 23% less power and achieves lower detection latency for traffic rates higher than 2.5 Gbit/sec.

the optimal values. Figure 8.9 shows the power consumption and detection latency of all approaches as a function of traffic rate. We see that LEoNIDS with both approaches consumes approximately the same power as the power-proportional NIDS, significantly lower than the consumption of the original Snort. Despite the lower consumption, LEoNIDS achieves a significantly lower detection latency than the power-proportional NIDS, close to the latency of the original system.

Space sharing performs quite better than time sharing: although both consume approximately the same power, space sharing achieves more than 40% lower detection latency. This is due to the non-preemptive priority queues used in time sharing: a high-priority packet may wait for a low-priority packet that is being processed. Moreover, the overall utilization of active cores in time sharing remain very high, so it cannot efficiently reduce the queuing delays of high-priority packets.

Overall, LEoNIDS with space sharing consumes 22% less power than the original system and it is able to detect attacks with an order of magnitude lower latency than the straight-forward power-proportional NIDS. Moreover, space sharing achieves a lower detection latency than the original system for rates higher than 2.5 Gbit/sec. This is due to the higher priority given at the beginning of each flow. As the original system does not give priority to these packets, it experiences higher de-

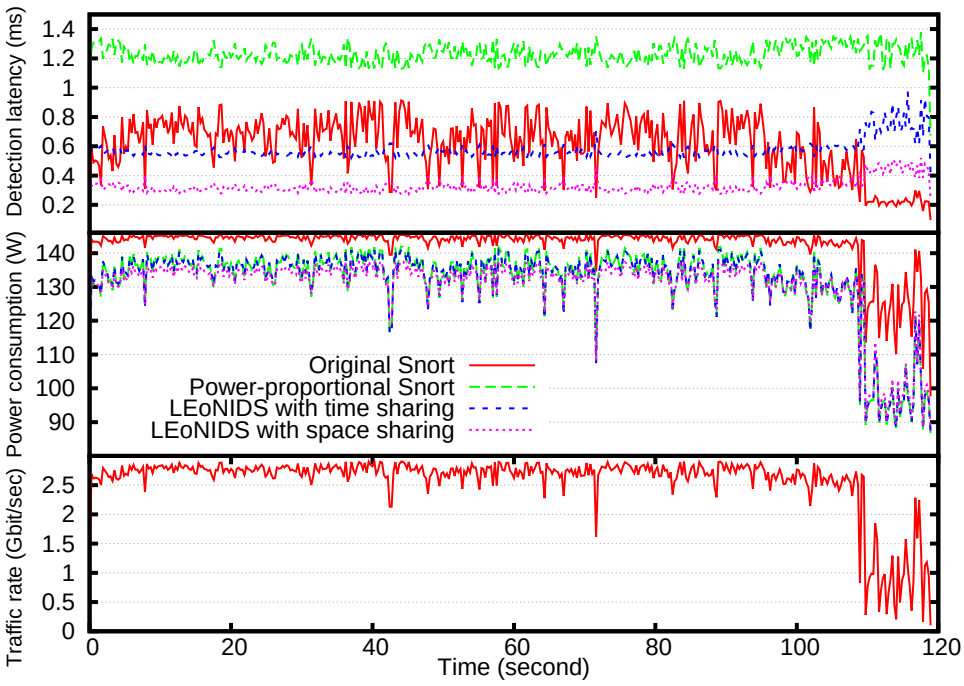


FIGURE 8.10: Space sharing performs better under realistic traffic variations. Traffic rate, power consumption, and detection latency over time.

tection latency compared to both time and space sharing at high traffic rates where all approaches result in an almost fully utilized system.

Realistic traffic variations. In our next experiment we compare all approaches in a realistic scenario of traffic variations. We replayed our one-hour long trace at its original rate using a 30x multiplier, resulting in an 120-seconds long experiment. Figure 8.10 shows the traffic rate, power consumption, and detection latency of all approaches over time. We see that again LEoNIDS with space sharing achieves the lowest detection latency among the other power-proportional approaches.

Active response. In our last experiment we examine how the detection latency of each approach affects the effectiveness of a NIDS reaction to actively terminate offending TCP connections. We configured Snort with *flexresp2* plugin for active response, and we added a rule to match a specific string and respond with reset to both source and destination hosts of the matched flow. While sending background traffic at 1.0 Gbit/sec, we were also sending connections with packets matching this string. We sent a constant number of packets per connection, while varying its duration. Figure 8.11 shows the percentage of successfully closed connections by active response when sending 100 such connections, as a function of connection's duration. We see that the straight-forward power-proportional NIDS cannot respond in time and close connections shorter than 6 ms with more than 50% probability. Contrary, LEoNIDS is able to terminate most connections lasting more than 3 ms, similar to the original Snort.

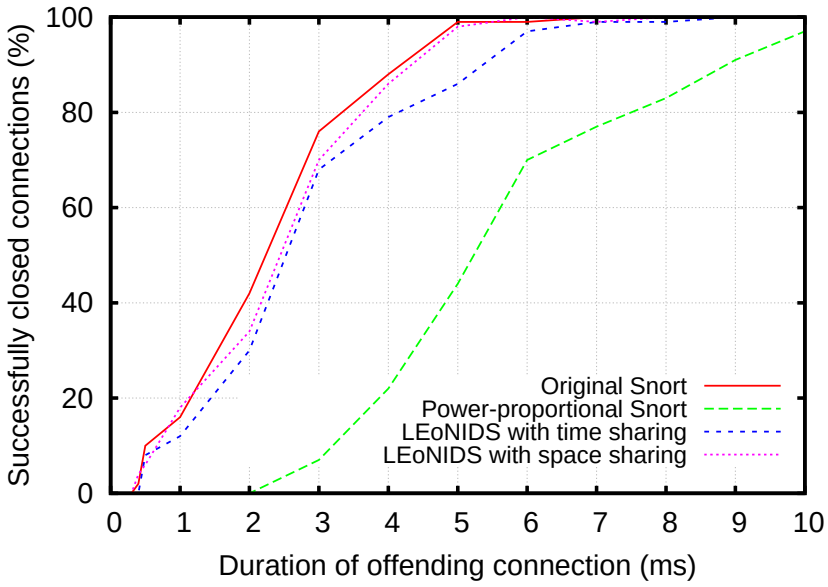


FIGURE 8.11: LEoNIDS close most offending connections longer than 3 ms. Percentage of closed connections using active response as a function of connection's duration.

8.1.6 Summary

In this work we studied the problem of improving the energy efficiency of a NIDS using common power management capabilities like DVFS and C-states. As NIDSs are usually overprovisioned to operate at the maximum link capacity, while these links are usually much less utilized, there are significant opportunities to reduce power consumption. However, while building a power-proportional NIDS, we identified an energy-latency tradeoff: the reduced power consumption results in a significant increase on the detection latency, which impedes a timely automatic reaction of the NIDS to the incoming attacks. By analyzing the detection latency we showed that the main reason for this increase is the high queuing delays imposed by the high core utilization.

We presented the design, implementation, and evaluation of LEoNIDS: a NIDS that resolves the energy-latency tradeoff. The key idea of LEoNIDS is to process with higher priority the first few bytes of each flow, which have a higher probability to carry an attack, to achieve low latency and fast detection. Then, we proposed two alternative techniques: time sharing and space sharing. Time sharing uses a typical priority queue scheduling, while space sharing uses dedicated cores with lower utilization to process high-priority packets. Our experimental evaluation shows that LEoNIDS performs better with space sharing than with time sharing. Overall, LEoNIDS with space sharing consumes significantly less power, proportionally to the load, and constantly low attack detection latency at the same time.

8.2 Long-Term Network Traffic Recording

Live traffic monitoring systems capture and process packets in real time. Regardless of the particular use, captured packets are usually discarded once processed. However, *recording* the raw network traffic to disk for long-term periods can be very useful for a multitude of applications, such as troubleshooting network problems and measuring traffic trends or observing the historical evolution of the traffic. Moreover, while the Internet evolves over the years, new applications and more security breaches appear. Thus, long-term recording of Internet traffic can significantly contribute to better analyze and understand the Internet evolution.

Network traffic recording is also critical for many security purposes. Anomaly detection techniques require a long-term baseline of past traffic to build profiles for normal traffic and users. Postmortem forensics analysis is also based on past traffic to identify malicious activities that happened before the time that an attack is detected. For instance, looking back in time can help us to identify how the attackers compromised a system, what they did, and find out which data have been exposed to them. Moreover, lawful interception and data retention have been enforced recently by many national regulations to enrich crime evidence by reconstructing past VoIP calls or other kinds of network-based communications.

When new vulnerabilities and attack signatures for Network Intrusion Detection Systems (NIDS) are released, long-term recording of network traffic allows to identify past attacks and compromised systems that otherwise would go undetected. Also, it is common practice to test new NIDS signatures using past traffic to eliminate false positives. NIDS and other passive monitoring applications are trained, tuned, and properly configured based on recorded traffic from the network in which they will be deployed. Packet traces are also commonly used for benchmarking network monitoring applications and can be replayed in different rates using tools like `tcpreplay` [145].

Unfortunately, recording all traffic in high volume networks is impossible even for short-term periods, due to the high storage needs. For instance, a network with 300 Mbit/sec average load requires about 3.2 TB of storage for recording one day's traffic. Thus, the limited storage resources of a commodity PC allow for storing hours or maybe a few days of traffic in the best case. However, recording the network traffic for long-term periods using a reasonable amount of storage would be extremely beneficial for all applications mentioned above.

Storing only the first few bytes from each packet, which typically corresponds to protocol headers, can reduce the required storage and increase data retention [96]. However, monitoring applications that need to inspect both the headers and the payload of the packets, a process widely known as *deep packet inspection* [60], cannot operate with header-only traces. Two traditional approaches for data reduction are aggregation and sampling. Aggregation is effective when the traffic's features of interest are known in advance, while sampling techniques select a representative group of packets uniformly over time. The sampling rate is an important parameter for the accuracy on inferring various network metrics. Higher sampling rates result

to better accuracy but require more storage space, and thus retention is reduced when using fixed-size storage. On the other hand, lower sampling rates increase data retention but inevitably reduce the accuracy of many applications.

In this chapter we present *RRDtrace*, a technique for storing packets for long-term periods in fixed-size storage, inspired by the popular RRDtool [104]. We choose to store full-payload packet traces, which provide a rich source of information suitable for all kinds of analyses, from coarse-grained measurements of network properties to fine-grained operations like deep packet inspection. *RRDtrace* divides time into intervals and retains more detail for more recent intervals, i.e, allocates more storage to recent time intervals and less storage to older time intervals. Also, older time intervals become longer than more recent ones. RRDtrace is based on an *aging* mechanism that dynamically reduces the space occupied by the data of a time interval as it ages, by keeping only a subset of the packets of that interval using sampling. Thus, as a time interval gets older, the sampling rate for storing its data decreases.

Many sampling techniques have been extensively studied for applications like traffic accounting, billing, and measurements like heavy-hitters identification and flow size estimation. However, the applicability of sampling techniques in other passive monitoring applications like traffic classification and intrusion detection has not received the same attention. Our study attempts to answer the following questions:

- *Which sampling strategies should be used to select a useful subset of packets when reducing the storage space that will allow us to infer as many as possible desirable properties from the trace? Which strategies are suitable for which properties?*

- *How much back in time can we go, i.e., what is the lowest sampling rate that still allows us to infer desirable properties from an RRDtrace with acceptable accuracy?*

To answer these questions, we evaluate the impact of three different sampling strategies by decreasing sampling rates on inferring desirable network properties using a large trace of real traffic. Our results indicate that RRDtrace using flow sampling can accurately estimate flow size distribution and distribution of flows among applications regardless of the sampling rate. Average flow size and percentage of traffic per application are estimated more accurately in recent time intervals. For estimating the percentage of malicious hosts and flows, reduction of traffic volume using a per-flow cutoff provides the more accurate estimates for recent intervals. Random packet sampling performs well only for few of the examined properties. Compared to a constant sampling rate strategy, RRDtrace can store traffic for arbitrary long time periods and offers higher accuracy for more recent traffic.

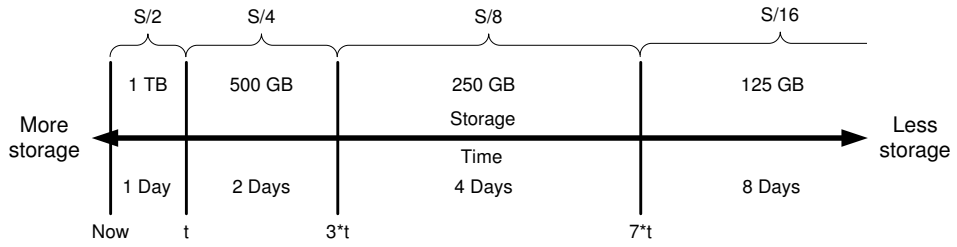


FIGURE 8.12: Storage allocation in RRDtrace for S=2 TB.

8.2.1 Our Approach: *RRDtrace*

Our approach, called *RRDtrace*, is inspired from the properties found in round robin databases. It aims to store full-payload packets for long-term periods in fixed-size storage. RRDtrace divides the time into unequal intervals and retains more packets from recent intervals, while keeping smaller subsets of packets from older intervals. Older time intervals are longer and utilize less storage. Recent time intervals are smaller with more storage assigned. The duration of time intervals and how the available storage is assigned to them can be defined either by the users, according to the network in which RRDtrace will be deployed, or automatically by RRDtrace.

A typical example of storage allocation in RRDtrace is shown in Figure 8.12. We assume that the available storage for RRDtrace is $S = 2$ TB. We select the initial time interval t_0 to be one day and we assign the half storage (1 TB) to it. The next time interval t_1 is twice as large as t_0 with the half storage of t_0, i.e., t_1 is two days long with 500 GB storage. Thus, in t_1 (days 2–3), 1 out of 4 packets that were initially stored is selected to remain in the trace. Each subsequent time interval is two times larger and has half the storage than its preceding one.

In this storage allocation algorithm different initial time intervals t_0 can be defined, occupying the half of the available storage. All the next intervals are formed based on t_0 and available storage S. In case that the traffic volume in t_0 is less than $S/2$, all packets in this interval can be stored. Else, packet sampling is imposed from the first time interval. An other option is to let RRDtrace to select the first interval t_0 in a way that all the packets during this interval are stored in the corresponding storage (with no sampling). Then, t_0 will be the time interval with traffic volume equal to $S/2$. This approach works well when the traffic volume in t_0 intervals does not vary significantly.

When a t_0 period passes, an *aging* daemon is responsible to appropriately reduce the storage used in each time interval. For instance, the number of packets stored during the last t_0 will be reduced by 25%, and similarly with the next intervals in order to conform with the storage allocation scheme described above. The aging daemon reduces the storage capacity in each interval by selecting a representative group of packets with the appropriate sampling rate. The packet selection strategy is an important parameter for the usefulness of RRDtrace.

We suggest the use of sampling instead of aggregation for two reasons. First, data is retained in the same format, which is very convenient for analysis and processing by existing applications. Moreover, aggregation requires knowledge of the traffic's features of interest in advance, whereas sampling allows the retention of arbitrary detail while at the same time reducing data volumes.

Sampling Strategies

Since RRDtrace may be used by multiple applications, different sampling strategies may be suitable for different applications. We have implemented three sampling strategies to evaluate their effectiveness using several monitoring applications. Each sampling strategy defines the way that k packets should be selected out of the total N packets in a time interval (sampling rate $s = k/N$), to respectively reduce the storage. We consider that a sampling rate has a similar effect in packets and storage reduction.

Packet Sampling The simplest strategy to select k out of N packets is systematic count-based sampling, i.e., selecting one every N/k packets. However, systematic sampling is vulnerable to bias errors due to synchronisation with periodic patterns in the traffic and can be predicted.

Random packet sampling avoids the potential problems of systematic sampling. We choose to implement stratified random sampling. In this technique, the N packets are divided to k equal groups (with size of N/k packets) and one packet from each group is randomly selected. In systematic count-based sampling the first packet of each group would be always selected.

Flow Sampling Research works by Hohn and Veitch [69] and Duffield et al. [48] have shown that packet sampling is inaccurate for the inference of flow statistics such as the original flow size distribution. For instance, it is easy to miss completely the short flows. Flow sampling has been proposed as an alternative to overcome the limitations of packet sampling. Hohn and Veitch [69] show that flow sampling improves the accuracy in flow statistics inference.

When a flow is selected, all the packets that belong to this flow are stored, while from an unselected flow no packets are stored. Flow sampling approaches for forming flow records focus mostly on selecting large flows, which has a larger impact to billing and accounting applications. So, non-uniform flow sampling techniques, like smart sampling [47] and sample-and-hold [52], have been proposed for accurate estimation of heavy hitters. These techniques give higher probabilities in large flows to be selected and form flow records.

In our case, we aim to select a representative group of flows for applications like traffic classification, building profiles, and security applications. Thus, we choose a uniform flow sampling approach. Random flow sampling with sampling rate s could be used. Similarly, hash-based sampling could be performed, using a hash function over the 5-tuple which defines a flow and then selects k out of

the possible N hash values. However, these approaches do not guarantee that the selected flows will result to k out of N packets selection, and to the desirable storage reduction, due to the heavy-tailed distribution of flow sizes. Therefore, hash-based and simple random flow sampling, as well as smart and sample-and-hold sampling strategies, cannot accurately reduce the storage.

We need to specify a flow sampling scheme that selects l flows out of the M total flows in a time interval, with k packets in total. This flow sampling scheme works as follows: First we classify packets into flows. During the classification, we maintain an indexing table with the flows sorted based on their size and a histogram with flow sizes. Then, we randomly select one flow at a time, with a size of x_i packets, while $\sum x_i < k$ stands. Only flows with size less than $k - \sum x_i$ packets that have not been selected so far, are candidates for selection. These flows can be easily found using the indexing table and the histogram with flow sizes. Assuming that we have F flows with size less than $k - \sum x_i$ packets that have not been selected before, a random number from 1 to F is used to select the corresponding flow from the indexing table. The selected flows are marked and removed from the indexing table and flow size histogram. The selection process ends when l flows with $\sum_{i=0}^{l} x_i = k$ have been selected. Finally, the packets from the selected flows are written to disk, with a second pass in the trace, in respect to the order that they have been received.

Per-Flow Cutoff Our third strategy for selecting representative packets is to use a per-flow cutoff, i.e., select always the first C packets of each flow. Time Machine [89] uses a statically user configured per-flow cutoff to limit the amount of traffic that will be stored. On the other hand, RRDtrace reduces the amount of traffic that will be stored according to the time interval that the traffic belongs to, thus different cutoffs are applied to different time intervals. As traffic ages, the per-flow cutoff will be properly reduced.

We implemented an algorithm that selects a per-flow cutoff C in a way that k packets are selected out of the total N packets. The algorithm is based on a histogram of aggregated statistics. In the first step we classify packets in flows. During this classification, we also maintain a table which indicates the number of flows that exceed each flow size. For instance, the position i of the table, $t[i]$, will contain the number of flows that have at least i packets. When the i_{th} packet of a flow is classified, $t[i]$ will be incremented by one.

Using this table, we can find the number of packets that correspond to a specific per-flow cutoff x from $\sum_{i=0}^{x} t[i]$. The selected cutoff C will be the largest position in the table that $\sum_{i=0}^{C} t[i] \leq k$ will be valid. In the second step of the algorithm, having the proper cutoff C, packets are classified again into flows and each packet is selected only if its position in the flow is less than C. Otherwise, the packet is not stored in the new file.

This per-flow cutoff strategy selects k packets in total from all the flows that appear in a time interval. Thus, it can accurately estimate the number of flows but

not their size. Its main advantage is that the trace will contain the first packets from all the flows, so it will be suitable for security applications, e.g., port scan and intrusion detection, but not for traffic classification and accounting applications.

Implementation

RRDtrace is implemented using two separate threads: the capture and aging daemons. The *capture daemon* uses libpcap [92] to capture packets for the t_0 interval, impose sampling, if needed, in t_0 and initially store the packets in a memory buffer. When the memory buffer becomes full, the packets are written to disk. Separate files are used for each t_0.

The *aging daemon* is responsible for reducing the storage as traffic ages. After each t_0, it reads packets from the files of each interval, imposes the new sampling rates and writes the selected packets to the updated files. The two threads do not access the disk concurrently to improve disk's performance. Thus, the aging daemon runs only when the capture daemon writes packets to the memory buffer.

Applications of RRDtrace

We focus on using RRDtrace for the following two classes of possible applications:

1) *Study the historical evolution of traffic:* Using RRDtrace we aim to infer the distribution of traffic among different applications, the distribution of flows sizes, the number of security alerts, the percentage of the malicious population and how all these change over the years.

2) *Security applications:*

   a) Building profiles for normal traffic patterns based on RRDtrace to be used by anomaly detection metrics.

   b) Forensics analysis, which often requires the reconstruction of past streams for lawful interception or inspecting past traffic from suspicious or compromised hosts to identify more malicious operations or sensitive data exposed to attackers.

   c) Intrusion detection in past traffic, for training new signatures to eliminate false positives, for detecting past attacks that were using a recently disclosed vulnerability, or for estimating the percentage of infected hosts.

8.2.2 Retention Study

We examine the operation of RRDtrace and compare its retention with other approaches by capturing and storing the traffic in the access link of an educational network. The average traffic load in the network is 178 Mbps with total traffic 1.92 TB/day on average. Assuming we have 2 TB available storage, t_0 should be set to

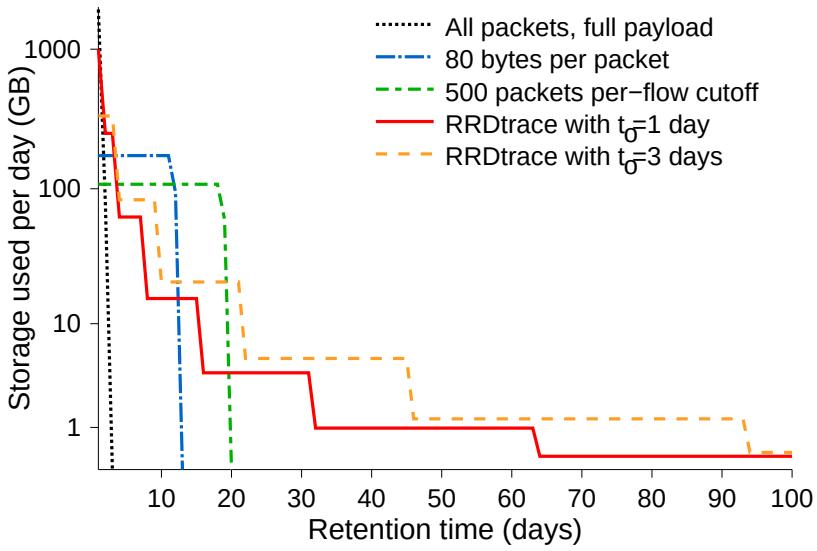


FIGURE 8.13: Retention time and storage utilization for RRDtrace and other approaches with 2 TB of available storage.

12.5 hours in order to store all packets during this interval in 1 TB. After t_0, for the next 25 hours, 25% of these packets will be stored in 500 TB.

Figure 8.13 presents the retention and the corresponding storage used per day for full-payload packet recording, headers-only recording (80 bytes per packet), when recording the first 500 packets per-flow and when using RRDtrace with $t_0 = 1$ day and $t_0 = 3$ days.

Since the daily traffic volume in the network is 1.92 TB, we can store all the packets with full payload for 25 hours only in the 2 TB storage. When capturing and storing only 80 bytes per packet, 173.22 GB are required per day, which results to 11.55 days retention. Applying a per-flow cutoff is a more effective approach, due to the heavy-tailed distribution of flow sizes. Using a cutoff of 500 packets per flow results to 107.76 GB/day stored and 18.56 days retention. A cutoff of 100 packets per flow results to 67.86 GB/day and 29.47 days retention.

On the other hand, retention in RRDtrace can be arbitrary large. Figure 8.13 shows the storage allocation in RRDtrace for the first 100 days using two different values of t_0. For $t_0 = 1$ day, 1 TB will be used for the last day's traffic, 15.6 GB/day for 8–15 days ago, 3.9 GB/day for 16–31 days and 976 MB/day for 32–63 days ago. Selecting 976 MB from the total 1.92 TB daily traffic implies 0.05% sampling rate. For one year ago, 15.3 MB/day traffic will be available, while for two years ago traffic 7.65 MB/day will be stored. When $t_0 = 3$ days, 333 GB/day will be used for the three last days. For 10–20 days ago, 20.83 GB/day will be stored, which implies 1.1% sampling rate. For one year ago, 81.4 MB/day traffic will be stored in this case.

8.2.3 Experimental Evaluation

To experimentally evaluate the usefulness of RRDtrace, we measure the accuracy of several properties when running passive monitoring applications and applying the different sampling strategies with decreasing sampling rates in a trace with real traffic. Our evaluation has three main objectives: First, to compare RRDtrace with uniform and constant sampling when both approaches reduce equally the size of the trace. Moreover, we aim to study how the three different sampling strategies with reducing sampling rates affect the accuracy on inferring traffic's properties from the RRDtrace. Finally, we examine how the accuracy is reduced across the retention time, as sampling rates are getting smaller, for different properties and sampling strategies.

We used an anonymized packet trace captured during one hour at the access link that connects an educational network with thousands of hosts to the Internet. The trace contains 73,162,723 packets, corresponding to 1,728,878 different flows, totalling about 46 GB in size.

In the first set of experiments, we compare RRDtrace with the three sampling strategies which use constant sampling rate, when all the approaches reduce the size of the trace to 10% of its original size, i.e., to $S = 4.6$ GB. Thus, we applied to the original trace packet and flow sampling with 10% sampling rate and per-flow cutoff of 74 packets per flow, which all resulted to 10% of the original trace's size. In RRDtrace, we used as t_0 the most recent $1/20$ interval of the trace. In this way, RRDtrace assigned the half of the available storage, $S/2$, to this interval, selecting all the packets from it. The next two more recent $1/20$ intervals of the trace were assigned $S/4$ storage, resulting to the selection of 25% of the packets during these intervals. For the four next intervals, 6.25% sampling rate was performed, and so forth. We tried all the sampling strategies with RRDtrace and we present results from the strategy that was found to perform better with each estimated property. The produced trace was always close to 4.6 GB, 10% of the initial trace's size. We report the accuracy of each measured property separately for each of the $1/20$ intervals of the trace, to compare the different approaches with RRDtrace.

For the second set of experiments, we applied packet sampling, flow sampling and per-flow cutoff to the original trace using sampling rates from 1 to $1/4096$, resulting in multiple sampled traces. For packet and flow sampling, where packets are selected in a random way, we produced 20 traces for each case and we present the average values. Thus, for each sampling rate we created traces with each strategy. Setting $t_0 = 1$ day, for each past day we plot the value inferred from the traces with the corresponding sampling rate. For instance, $1/4096$ sampling rate corresponds to 64–127 days ago. In sampling rates below $1/256$, the respective per-flow cutoff becomes 1 packet per flow, which means that this strategy cannot be applied in very low sampling rates.

We first evaluate the accuracy on estimating flow statistics, like the original flow size distribution and average flow size. Then, we examine the accuracy on inferring the distribution of traffic and flows among the applications that generate

them, using the Appmon tool [12]. Snort NIDS [124] was used to examine if the percentage of hosts and flows that produce security alerts can be inferred from the sampled traces. The accuracy for each property is measured by comparing the value inferred from each sampled trace with the real value from the unsampled trace. In the remaining of this section, we present the evaluation for each property separately.

Flow Size Distribution

Flow size distribution is a useful metric for traffic engineering, traffic classification, anomaly detection techniques and for studying how network traffic changes over the time. We examine the accuracy on inferring the average and mean flow size and the original distribution of flow sizes using sampled traces.

Figure 8.14 compares the accuracy on the estimation of average flow size using RRDtrace and sampling with constant rate for the reduction of the trace's size to 10% of its original size. RRDtrace used flow sampling, that is the best choice for inferring flow statistics, with adaptive sampling rate to retain more packets from the first parts of the trace and result also to 10% of the original trace's size. Packet and flow sampling used 10% sampling rate, while 74 packets per-flow resulted to the same reduction in the trace's size. We plot the accuracy of the average flow size estimation separately from the most recent to the older $1/20$ time interval of the original trace, by comparing with the actual value from the respective interval in the unsampled trace.

During the most recent interval RRDtrace retains all the packets, thus it is 100% accurate. For the next two intervals RRDtrace performs 25% flow sampling, so it is more accurate than 10% flow sampling. In the next four intervals, RRDtrace uses 6.25% sampling rate and its accuracy remains close to 10% flow sampling. Overall, compared with constant flow sampling, for the three more recent intervals RRDtrace is more accurate, for the next four intervals with similar accuracy and in the older intervals flow sampling outperforms RRDtrace from 5% up to mostly 20%. If we need to further reduce the storage size, e.g., to 1% of the original size using RRDtrace and 1% sampling rate, RRDtrace will be more accurate for a longer time period. When RRDtrace is used for live traffic recording, the dynamic storage re-assignment is the only way to retain data for arbitrary long periods, since sampling with constant rate will have limited retention using fixed-size storage.

Figure 8.15 compares the accuracy of the three different sampling strategies in RRDtrace for estimating average flow size. Flow sampling is always more accurate than packet sampling and per-flow cutoff, which are not effective strategies for the inference of flow sizes. Using flow sampling, average flow size is accurately estimated for a few days period. For the past two days, the estimation is 96.5% accurate with sampling rate 25%. RRDtrace slightly overestimates the average flow size, due to more possibilities for the selection of very large flows. For 4–7 days ago, the estimation is 94.7% accurate. RRDtrace tends to underestimate the average flow size for lower sampling rates. Due to the heavy tailed distribution of

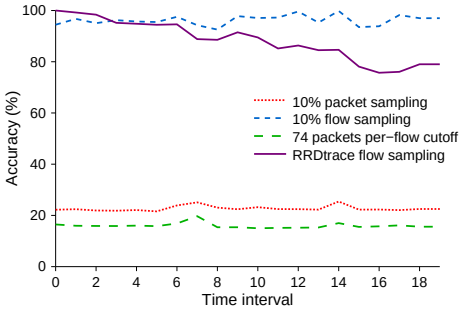


FIGURE 8.14: Accuracy on average flow size estimation using RRDtrace and a constant 10% sampling rate.

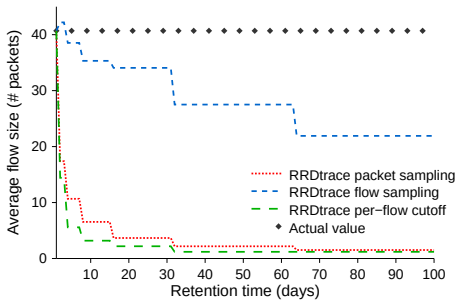


FIGURE 8.15: Average flow size estimation using RRDtrace with different sampling strategies.

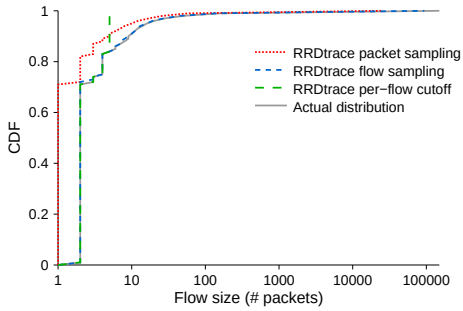


FIGURE 8.16: Flow size distribution for 8–15 days ago, with an $1/64$ sampling rate and 5 packets per-flow cutoff.

flow sizes, there is a higher possibility for small flows to be randomly selected in low sampling rates than large flows. Moreover, very large flows cannot be selected in low sampling rates due to the storage size limitation. Up to one month ago, average flow size is estimated with 83.7% accuracy.

On the other hand, flow size distribution can be accurately estimated using flow sampling with low sampling rates. Figure 8.16 shows the cumulative distribution of flow sizes for packet and flow sampling with $1/64$ sampling rate and for 5 packets per-flow cutoff. Packet sampling is not accurate, since many small flows are completely lost. Per-flow cutoff strategy can estimate correctly the size of flows up to 5 packets in this case, according to the cutoff limit. The rest of the flows are considered with 5 packets size and there is no clue for their actual size. Flow sampling is accurate even with $1/4096$ sampling rate. Thus, in flow size distribution property the accuracy does not depend on the sampling rate for flow sampling. Flow sampling and per-flow cutoff estimate correctly the mean flow size, that is 2 packets per flow, while packet sampling incorrectly estimate it as one packet per flow.

While per-flow cutoff cannot accurately estimate the flow sizes, it accurately estimates the actual number of flows, since it retains at least one packet from each flow. Our flow sampling strategy can provide a less accurate estimation of the actual number of flows. If we had chosen to select exactly $s \times M$ flows from an interval with M total flows for a sampling rate s, we could infer the actual number of flows by multiplying with s the flows found in the sampled trace. Since this strategy does not always reduce the storage by s, we chose to select the number of flows with the desirable reduction in storage. Even so, we observe that for high sampling rates the chosen flow sampling strategy selects about $s \times M$ flows, so it can infer the actual number of flows. For low rates, it tends to select more than $s \times M$ flows and thus overestimates the actual number of flows up to two times in $1/4096$ sampling rate.

Per-Application Traffic Classification

The next property we examine is the classification of network traffic and flows to the applications that generate them. We aim to infer the percentage of traffic and flows that each application contributes to the total traffic and flows in the network. For these measurements we ran Appmon with our sampled traces. Appmon classifies flows and traffic into applications using both port-based classification and deep packet inspection to identify peer-to-peer and multimedia applications that use dynamically allocated port numbers, based on application specific signatures. For instance, Web traffic is all the packets from/to port 80, except from peer-to-peer packets masqueraded as Web packets. A flow is classified as BitTorrent flow when a packet of the flow, usually the first, contains a BitTorrent protocol-specific string. Specifically, the Peer Wire Protocol in BitTorrent establishes a handshake using well known keywords in the first packets. BitTorrent traffic is all the packets that belong to a flow classified as BitTorrent. We present the results for the two most popular applications found in the trace, Web and BitTorrent.

In Figures 8.17 and 8.20 we compare the accuracy of RRDtrace with the accuracy of 10% packet and flow sampling and 74 packets per-flow cutoff on estimating the percentages of Web and BitTorrent traffic respectively. In case of Web traffic percentage, packet sampling is clearly the most accurate strategy, due to the simple port-based classification. However, packet sampling significantly affects the detection of BitTorrent traffic, so flow sampling is the most accurate approach in this case. Comparing RRDtrace with constant 10% flow sampling in Figure 8.20, we observe the effect of sampling rate adaptation in RRDtrace algorithm. For the three most recent intervals RRDtrace is clearly more accurate, for the next four intervals almost equal and for the rest of the trace provides less but close accuracy compared with constant flow sampling.

Figures 8.18 and 8.21 compare the different sampling strategies across the retention time for Web and BitTorrent traffic percentage estimation respectively. We observe that packet sampling provides very accurate estimates of the Web traffic percentage regardless of how old the data are, i.e., how low the sampling rate is.

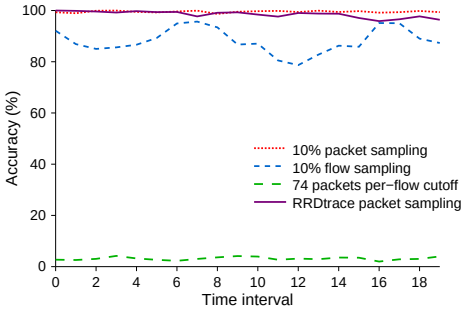


FIGURE 8.17: Accuracy on Web traffic percentage estimation using RRDtrace and a constant 10% sampling rate.

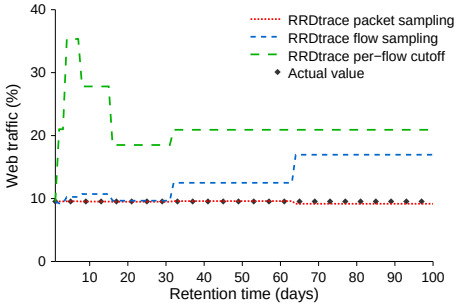


FIGURE 8.18: Percentage of Web traffic using RRDtrace with different sampling strategies.

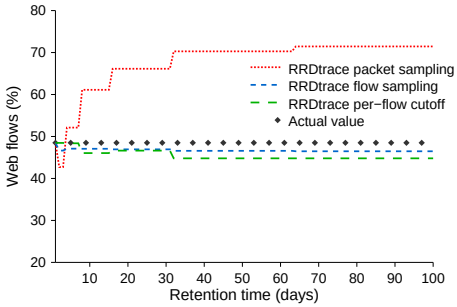


FIGURE 8.19: Percentage of Web flows out of the classified flows.

Thus, packet sampling fits well with the simple port-based traffic classification. However, packet sampling cannot estimate accurately the percentage of BitTorrent traffic even for recent traffic with higher sampling rates. This is because packet sampling affects significantly the detection of a BitTorrent flow. Since packets are randomly selected, the packet which contains the BitTorrent keyword may be missed. As a consequence, all the selected packets from this flow will not be classified as BitTorrent packets, leading to significant error in the estimation's accuracy.

On the other hand, flow sampling is the most accurate approach for estimating the percentage of BitTorrent traffic. In flow sampling, all packets from a selected flow are present, so the flow-based classification process is not affected. Thus, it can estimate the percentage of BitTorrent traffic till 30 days ago with more than 87% accuracy. For the same period, it can estimate the Web traffic percentage with accuracy 98.75%. The decreasing sampling rates affect the accuracy of flow sampling. While it has good accuracy up to 30 days ago, for older time periods it tends to overestimate Web traffic and underestimate the BitTorrent traffic percentage.

The third packet selection strategy, based on a per-flow cutoff, cannot account correctly the percentage of traffic for each application. While most of the traffic

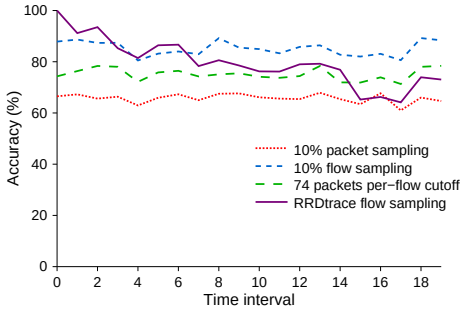


FIGURE 8.20: Accuracy on BitTorrent traffic percentage estimation using RRDtrace and a constant 10% sampling rate.

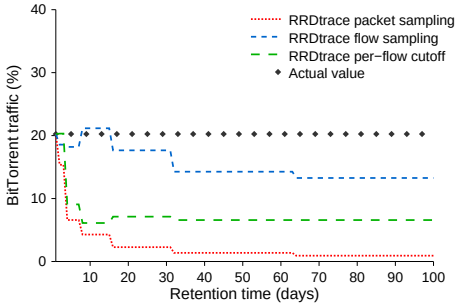


FIGURE 8.21: Percentage of BitTorrent traffic using RRDtrace with different sampling strategies.

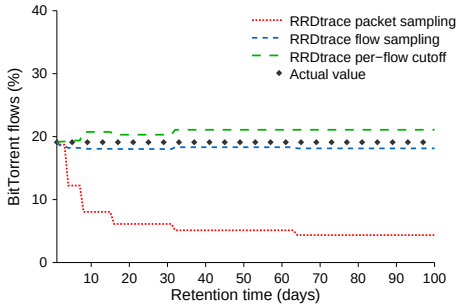


FIGURE 8.22: Percentage of BitTorrent flows out of the classified flows.

can be successfully classified by the first packets of the flow, the cutoff affects non-uniformly the traffic volume stored from each application. For instance, BitTorrent has usually large flows which are highly affected from the cutoff, resulting to an underestimation of the BitTorrent traffic percentage, as we observe in Figure 8.21. On the other hand, Web flows are typically smaller and thus less affected, leading to an overestimation of Web traffic.

However, per-flow cutoff can accurately estimate the number of Web and BitTorrent flows, even if it cannot infer the correct percentages over the total traffic. Even with one packet per flow, BitTorrent flows can be usually detected. Figure 8.19 shows the percentage of Web flows and Figure 8.22 the percentage of BitTorrent flows out of the flows that were successfully classified. We observe that both flow sampling and per-flow cutoff can accurately estimate the percentage of flows for each application for arbitrary low sampling rates, with flow sampling be slightly more accurate. While packet sampling can estimate successfully the Web traffic percentage, it cannot estimate correctly the number of Web flows.

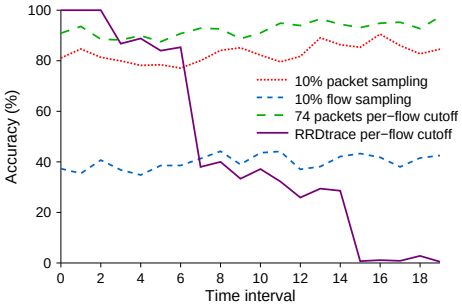


FIGURE 8.23: Accuracy on malicious hosts percentage estimation using RRDtrace and a constant 10% sampling rate.

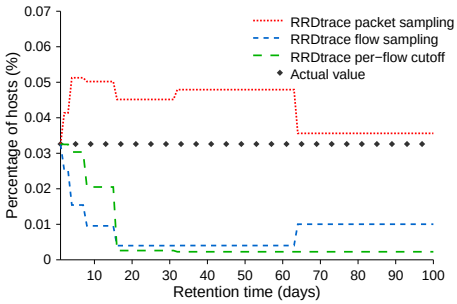


FIGURE 8.24: Percentage of hosts that trigger security alerts.

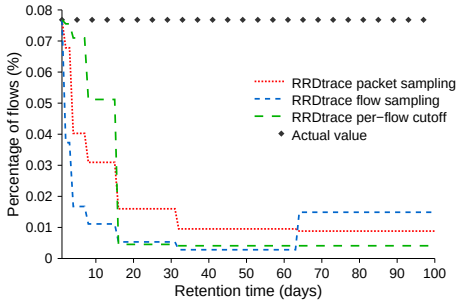


FIGURE 8.25: Percentage of flows that trigger security alerts.

Estimation of Malicious Population

Network traffic stored in sampled traces can provide some information about past networking attacks, suspicious activities and malicious hosts. Instead of trying to infer the actual attacks that happened in the past, we focus on estimating the percentage of hosts and flows that generates security alerts and how this percentage changes over the time. We ran Snort NIDS in the unsampled and sampled traces with each sampling strategy and sampling rate, aiming to measure the accuracy on estimating these percentages. We consider the source IP addresses of packets that produce Snort alerts as malicious hosts. The percentage of malicious hosts is estimated by dividing the number of unique malicious hosts with the total number of hosts that are present in each sampled trace. We also ran Snort with the reduced trace in 10% of its original size with RRDtrace and constant sampling techniques and compare their accuracy on the estimated percentage of malicious hosts.

We observe that with a per-flow cutoff of 74 packets, 84% of the alerts that were triggered in the original trace are also detected. For security applications, per-flow cutoff is a good choice for data reduction since a large class of attacks is detected in the beginning of the flows. For instance, network service probes, brute force login

attempts and code-injection attacks usually appear in the first few hundred packets of a network flow. Moreover, due to the heavy tailed distribution of flow sizes, the 74-packet per-flow cutoff affects only 1.6% of the flows that contribute most of the traffic, resulting to a reduction rate of 90%.

Figure 8.23 presents the results for the reduced trace. Per-flow cutoff strategy has the best accuracy in estimation of the malicious host percentage. Packet sampling has better accuracy than flow sampling on the malicious host estimation. Using the trace sampled with 10% packet sampling, Snort finds significantly more attacks than with the trace produced with 10% flow sampling, 18% and 3% of the actual alerts respectively. Per-flow cutoff strategy retains all the hosts that were present in the original trace, while packet sampling only a subset of them (27.5%). Thus, the accuracy of the malicious hosts percentage in case of packet sampling is close to the accuracy when using the per-flow cutoff.

Flow sampling is not a good choice for this property. Therefore, for estimating the malicious hosts percentage we use the per-flow cutoff strategy with RRDtrace, which dynamically adapts the cutoff to store more packets per flow for the recent time intervals. In the three more recent intervals RRDtrace has 100% accuracy, since the cutoff of 2754 packets per flow that is applied in second and third intervals does not affect the malicious hosts detection. After the seventh interval, the accuracy of RRDtrace with per-flow cutoff degrades significantly, since the 5 packets per flow cutoff results to less malicious hosts be detected.

Figure 8.24 shows the effect of sampling rates on each strategy when estimating the percentage of malicious hosts. Per-flow cutoff is very accurate for 7 days ago and has reasonable accuracy till 15 days in the past. For older traffic, the lower sampling rates affect significantly its accuracy, leading to underestimation of malicious hosts percentage. This happens because this strategy retains all the hosts in the trace but less packets from each one, so less attacks and malicious hosts will be detected at lower sampling rates, resulting to a reduced percentage. On the other hand, packet sampling overestimates the percentage of malicious hosts and its accuracy is not affected by decreasing sampling rates. With the reduction of sampling rate, both the number of detected malicious hosts and the number of total hosts in the sampled trace are reduced. Therefore, while per-flow cutoff is more accurate at high sampling rates, at lower sampling rates packet sampling provides best accuracy and should be preferred.

Figure 8.25 presents the percentage of flows that trigger alerts in Snort out of the total flows found in the sampled traces. As we expected, per-flow cutoff provides the most accurate estimations for the last 15 days, but for older time periods it degrades significantly. All the sampling strategies are highly affected by the reducing sampling rates in this case. With packet sampling Snort finds a reasonable number of alerts and malicious flows, but the total number of flows found in the sampled trace is not proportional with the sampling rate. While flow sampling selects the proper amount of flows, less alerts are found in the produced traces compared with packet sampling and the percentage of malicious flows is underestimated.

8.2.4 Summary

Recording raw network traffic for long-term periods is extremely useful for a multitude of monitoring and security applications, such as troubleshooting network problems, studying the Internet evolution, postmortem forensics analysis and estimating the malicious population over time. The high volumes of network traffic highlight the need for data reduction and optimized traffic storage systems. In this paper we present RRDtrace, a technique that enables storing raw network packets in fixed-size disk space for arbitrary long periods, while retaining more detailed information for most recent traffic. RRDtrace dynamically reduces storage space as traffic ages using three alternative sampling strategies: packet sampling, flow sampling, and per-flow cutoff.

We experimentally evaluated RRDtrace with each sampling strategy by measuring the accuracy of flow size distribution estimation, traffic classification, and malicious hosts detection across the retention period using real traffic. Our main findings are the following:

1. When RRDtrace is used offline to reduce the size of a trace, it provides higher accuracy for the most recent part and the same accuracy for the rest, compared to constant sampling that has the same effect in trace size. When RRDtrace is used for live recording, it can store packets for arbitrary long periods based on the dynamic storage reduction, while constant sampling has limited retention.

2. Some properties can be accurately inferred regardless of how old the traffic is, i.e., using arbitrary low sampling rates. Such properties include flow size distribution using flow sampling, the percentage of Web and BitTorrent flows using flow sampling or a per-flow cutoff, and the percentage of Web traffic using packet sampling.

3. In contrast, other properties are highly affected by sampling rate and can be accurately inferred only in recent periods. Such properties include average flow size, percentage of BitTorrent traffic, and percentage of hosts and flows that trigger security alerts.

4. Flow sampling is overall the most robust technique for flow statistics and traffic classification inference, but it performs poorly in estimation of malicious population.

5. Per-flow cutoff strategy can estimate the actual number of flows and detect more attacks. However, it is not able to infer flow size and cannot be used with low sampling rates, as the cutoff reaches rapidly to one packet per flow and cannot be further reduced.

6. Packet sampling can estimate very accurately the percentage of Web traffic but not BitTorrent traffic, as it highly affects the corresponding detection algorithm. Moreover, it cannot estimate correctly any flow statistics.

7. To estimate the percentage of malicious hosts, per-flow cutoff is more accurate for recent time intervals, while packet sampling is more accurate for older time intervals with low sampling rates.

9 Conclusions

9.1 Summary

In this dissertation we explored the problems that arise when building network traffic monitoring systems under heavy load. We showed the need to improve performance and robustness of monitoring systems in a generic way transparently from the applications. More specifically, we showed that we need to improve memory access locality to achieve better performance (Chapter 4), the necessity for overload control mechanisms in network monitoring systems (Chapter 5), and defences against overload attacks (Chapter 6). Moreover, we demonstrated the need to use the proper abstractions in network monitoring frameworks for improved performance and expressiveness (Chapter 7). Finally, we showed that similar approaches can be used for two other applications (Chapter 8): reducing the detection latency in energy-efficient NIDS and long-term traffic recording.

Towards all these goals, we followed the approach of amplifying the core components of a network monitoring system with intelligence based on transport-layer information. We demonstrated that such intelligence can improve code and data memory locality, provide efficient overload control mechanisms, reduce the detection latency of NIDS, and lead to overall improvements in performance and accuracy of monitoring applications. Moreover, we rely on abstractions derived from transport-layer to build new frameworks that facilitate the development of emerging network monitoring applications with high performance under heavy load.

The main contributions of this work are the new techniques we proposed for efficient traffic monitoring under heavy load. We designed and implemented these techniques, either within existing libraries or by introducing new frameworks, and we evaluated their efficiency while exploring their main properties and existing tradeoffs.

First, we studied memory locality in network monitoring applications and we proposed *locality buffering* to improve code and data locality, reduce L2 cache misses, and improve performance in a transparent way to the applications (Chapter 4). Then, we focused on overload control. We proposed *selective packet discarding* technique to gracefully adapt to overload conditions by pro-actively discarding the less important packets for NIDS, which are the packets towards the end of large flows (Chapter 5). To tolerate overload attacks, such as algorithmic complexity or denial of service attacks, we proposed *selective packet paging*: a two-layer memory management system to store excess packets in secondary storage systems, accompanied with a randomization-based approach to detect crafted packets attacking the monitoring system (Chapter 6).

To enable the development of efficient network monitoring at the transport layer and beyond, we proposed the *Stream capture library (Scap)* (Chapter 7). Scap delivers reassembled transport-layer streams and an expressive API for stream-oriented traffic analysis based on the Stream abstraction. It also offers a variety of features, such as stream truncation at kernel or NIC level (subzero copy), prioritized packet loss, best-effort stream reassembly, and inherent support for multicore architectures.

We also applied similar ideas to solve two other problems in network monitoring systems (Chapter 8). First, we exposed the energy-latency tradeoff in intrusion detection systems: as we reduce power consumption with state-of-the-art approaches, such as frequency scaling and core deactivation, we noticed a significant increase in the detection latency. We showed that this impedes a timely automatic reaction to the incoming attacks, e.g., using an active response to terminate offending TCP connections. To resolve the energy-latency tradeoff, we presented LEoNIDS: a low-latency and low-energy NIDS. Low energy is achieved by state-of-the-art power management techniques tailored for NIDS, while low-latency is achieved by assigning priorities to the captured packets (Section 8.1). We assign higher priority to the packets that are more likely to contain an attack, which are the few first packets of each flow, and we showed that using dedicated cores to process these high-priority packets (*space sharing*) is more efficient that using a typical priority queue per each core (*time sharing*).

Finally, we presented a new technique for archiving network traffic for long-term periods using fixed-size storage (Section 8.2). This technique, called *RRD-trace*, is based on *traffic aging* idea: we keep more traffic for recent time intervals, and we reduce the traffic that remains on disk as it gets older. To select smaller representative samples as time passes, we explored different sampling strategies: packet sampling, flow sampling, and flow size cutoff. In packet and flow sampling, we randomly select packets or flows that will be retained, based on the sampling rate. In flow size cutoff, we keep less bytes per each flow as the flow becomes older. We studied the effect of these strategies on the accuracy of common network monitoring applications, and we found that the best strategy depends on the application's needs: while some applications need to retain less bytes for more flows, other applications need to retain more bytes for less flows.

Our work resulted in several prototype implementations: (i) a modified libpcap with locality buffering, (ii) a Snort preprocessor with selective packet discarding, (iii) a modified libpcap with selective packet paging, (iv) the Stream capture library (Scap) for stream-oriented traffic capture and analysis, consisting of a kernel module, a user-level library, and few sample applications, (v) a new kernel module, a modified libpcap, and modifications into the Snort NIDS for an energy-efficient NIDS with low detection latency, and (vi) the RRDtrace tool for long-term network traffic recording. We used these prototype implementations for experimental evaluation in our studies. Moreover, we plan to release the libraries and tools we developed or modified so that they can be used by researchers or network operators interested in efficient network traffic monitoring.

Our experimental evaluation results showed that the techniques we proposed and their prototype implementations are able to improve the efficiency of network traffic monitoring systems, offering effective overload control mechanisms and tolerance against sophisticated evasion attempts and overload attacks. Moreover, we showed improved memory locality and improved energy-efficiency without affecting other important performance metrics like throughput and latency.

9.2 Future Work

We believe that as network traffic and Internet applications become more complex, with an increasing number of security incidents, we will need more effective network monitoring to ensure the correct and secure operation of today's and future networks. As network traffic volume increases, monitoring applications become more complex, and attackers more sophisticated, we will always need to improve the existing network monitoring frameworks, which should also adapt to the new needs of the monitoring applications and utilize advances in commodity hardware. Towards this need, we believe that our work contributes to improve the state-of-the-art in network monitoring research directions we explored in this dissertation. As this area is rapidly evolving, due to the dynamic nature of the Internet traffic, applications, and attacks, along with the frequent advances in commodity hardware, such efforts should be continued in the future.

There are several directions that can be explored as part of future work in the area of network monitoring and security. To improve the performance and efficiency of network monitoring systems, routing-assisted techniques can be developed. For instance, adding intelligence into a router for selectively forwarding only the most interesting traffic for network monitoring or security analysis (e.g., the first few bytes of each flow) to a passive monitoring system may improve the overall performance of the network.

Moreover, recent advances in Software-Defined Networking (SDN) and the use of OpenFlow switches can be explored to improve network monitoring systems. For instance, network monitoring can benefit from SDN by collecting new sources of monitoring data from OpenFlow switches. Another direction could be to

dynamically route the network traffic using SDN principles in a way that it passes through the proper network monitoring sensors or intrusion detection systems, ensuring correct traffic inspection and avoid overloaded monitoring systems or other middleboxes. Also, SDN applications may benefit from network monitoring data and applications as well. For instance, getting input data from monitoring probes may help an SDN routing architecture to make the proper routing decisions. Thus, coupling SDN with network monitoring systems may be helpful for both worlds.

In future work, we could also explore additional directions related to our proposed techniques. We can exploit the memory access locality we observed in this work for more efficient traffic processing in multicore processors. More specifically, we can improve the performance of multicore systems by splitting the traffic in a way that related packets (not only packets belonging to the same connection) will be scheduled for processing by the same core. For example, packets belonging to the same high-level application and will result in a similar processing by the monitoring application can be processed by the same core to exploit memory locality and optimize performance.

Moreover, we can integrate our proposed techniques with other approaches, such as zero copy APIs and other high-performance packet capture and processing systems, to combine their performance benefits. We can also use selective packet paging with faster disk systems (e.g., SSD disks, RAID or other very fast storage systems) for improved performance, and utilize other advances in storage area.

Similarly, we can implement the RRDtrace tool with indexing approaches for faster query responses and optimized storage systems to improve the throughput that packets are stored to disk. Among the future work with RRDtrace will be to find an implement a single sampling strategy to reduce data based on traffic aging. Also, our future work may include building real-world applications using Scap.

To improve the query response times for retrospective analysis in network traffic recording systems, more sophisticated indexing techniques can be designed and implemented. Moreover, exploiting locality can also significantly improve the query response times of traffic recording systems: by properly rearranging the packet stream before writing packets to disk we can place related packets, which may be accessed by the same queries, in adjacent positions in disk. This way, disk accesses can be localized so that the typically low disk access speed can be significantly increased. Especially when this packet reordering is combined with packet indexing techniques, the disk accesses can be localized to a specific part of the disk, where most packet matching the same query can be found, while the larger part of the trace will not be read. Such locality can be achieved with some knowledge about the common queries, e.g., based on common header fields or application protocols. Thus, packets can be sorted before they are stored in disk based on this fields.

In future work, we can also explore how we can leverage the recent advances in cloud computing to improve network traffic monitoring. Cheap processing power and storage offered by cloud providers today might be utilized by monitoring and security systems to move network traffic processing and storage towards such cloud

systems. Moreover, the popularity of cloud computing in a multitude of applications today require to build network monitoring systems tailored for cloud computing systems and applications. Our proposed ideas could be utilized in such cloud network monitoring systems as well, due to the heavy traffic load in data centers. Thus, selective packet processing approaches can reduce the cost and at the same time improve the performance and accuracy in such environments. Moreover, we suggest to develop overload control mechanisms and defenses again evasion attempts and overload denial of service attacks. Especially when network traffic may be transfered from a monitoring sensor to another processing unit (in a local or remote network), in a distributed network monitoring system, a selective transfer of the captured traffic will reduce the bandwidth overhead as well.

Finally, a big challenge in network traffic monitoring and network security systems today is the inspection and analysis of encrypted traffic. Application protocols using encryption, like HTTPS, become increasingly popular in the Internet today. Although this increase significantly the security of the end users and of these applications, encryption may be also employed by malicious users and malicious software to hide their activities and avoid detection by network-level security systems. This poses significant challenges to network monitoring systems and NIDS. Therefore, these systems need to evolve and develop new analysis techniques to address this issue. For instance, behavioral analysis using machine learning techniques can be combined with traditional signature-based traffic inspection to recognize malicious or other interesting activities in the encrypted network packets, and classify them into their respective high-level applications. Another alternative is to deploy SSL proxies in networks that need to inspect the unencrypted SSL network traffic for security or other important purposes. This way, traffic will not be encrypted at the monitoring system, which has also an SSL proxy, but it will be encrypted in the rest of the network path. However, the deployment of such SSL proxies is not always possible, while SSL traffic is a (popular) subset of the existing encrypted traffic in the Internet.

Bibliography

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

[2] Dag ethernet network monitoring cards, Endace measurement systems. http://www.endace.com/.

[3] Fprobe: NetFlow probes. http://fprobe.sourceforge.net/.

[4] Intel Unveils Next Gen SSDs with 6Gbit/sec Throughput. http://www.computerworld.com/s/article/9211819/Intel_unveils_next_gen_SSDs_with_6Gbit_sec_throughput.

[5] Libnids. http://libnids.sourceforge.net/.

[6] Performance application programming interface. http://icl.cs.utk.edu/papi/.

[7] Sourcefire vulnerability research team (vrt). http://www.snort.org/vrt/.

[8] Y. Afek, A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. MCA2: Multi-Core Architecture for Mitigating Complexity Attacks. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2012.

[9] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18, 1975.

[10] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, M. B. Greenwald, and J. M. Smith. Efficient Packet Monitoring for Network Management. In *IFIP/IEEE Network Operations and Management Symposium (NOMS)*, 2002.

[11] E. Anderson and M. Arlitt. Full Packet Capture and Offline Analysis on 1 and 10 Gb/s Networks. Technical report, HP Labs, 2006.

[12] D. Antoniades, M. Polychronakis, S. Antonatos, E. P. Markatos, S. Ubik, and A. Oslebo. Appmon: An Application for Accurate per Application Traffic Characterization. In *IST Broadband Europe 2006 Conference*, 2006.

[13] M. Attig and J. Lockwood. A Framework for Rule Processing in Reconfigurable Network Systems. In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2005.

[14] Z. K. Baker and V. K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2004.

[15] M. Bando, N. Artan, and H. Chao. Scalable Lookahead Regular Expression Detection System for Deep Packet Inspection. *IEEE/ACM Transactions on Networking*, 20(3), 2012.

[16] P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, D. Amores-López, and J. Solé-Pareta. Load Shedding in Network Monitoring Applications. In *USENIX Annual Technical Conference (ATC)*, 2007.

[17] A. Biswas and P. Sinha. On improving performance of Network Intrusion Detection Systems by efficient packet capturing. In *IFIP/IEEE Network Operations and Management Symposium (NOMS)*, 2006.

[18] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi. On Multi—gigabit Packet Capturing with Multi—core Commodity Hardware. In *International Conference on Passive and Active Measurement*, 2012.

[19] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[20] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina. Impact of Packet Sampling on Anomaly Detection Metrics. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2006.

[21] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle. Comparing and Improving Current Packet Capturing Solutions Based on Commodity Hardware. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2010.

[22] T. Bray. Bonnie Benchmark for UNIX Filesystem Operations. `http://www.textuality.com/bonnie/`.

[23] J. B. D. Cabrera, J. Gosar, W. Lee, and R. K. Mehra. On the Statistical Distribution of Processing Times in Network Intrusion Detection. In *IEEE Conference on Decision and Control*, 2004.

[24] M. Canini, D. Fay, D. J. Miller, A. W. Moore, and R. Bolla. Per Flow Packet Sampling for High-Speed Network Monitoring. In *International Conference on Communication Systems and Networks (COMSNETS)*, 2009.

[25] S. Chandrasekaran and M. Franklin. Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *International Conference on Very Large Data Bases (VLDB)*, 2004.

[26] Chelsio Communications. T4 unified wire adapters. `http://www.chelsio.com/adapters/t4_unified_wire_adapters`.

[27] T.-H. Cheng, Y.-D. Lin, Y.-C. Lai, and P.-C. Lin. Evasion Techniques: Sneaking through Your Intrusion Detection/Prevention Systems. *IEEE Communications Surveys Tutorials*, 14(4), 2012.

[28] B.-Y. Choi, J. Park, and Z.-L. Zhang. Adaptive random sampling for load change detection. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2002.

[29] Cisco Systems. Cisco IOS Netflow. `http://www.cisco.com/warp/public/732/netflow/`.

[30] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kone, and A. Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In *Workshop on Network Processors and Applications*, 2004.

[31] E. Cooke, A. Myrick, D. Rusek, and F. Jahanian. Resource-Aware Multi-Format Network Security Data Storage. In *SIGCOMM Workshop on Large Scale Attack Defense (LSAD)*, 2006.

[32] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a Stream Database for Network Applications. In *ACM SIGMOD International Conference on Management of Data*, 2003.

[33] S. Crosby. Denial of Service Through Regular Expressions. *USENIX Security Symposium*, 2003.

[34] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security Symposium*, 2003.

[35] W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos. SafeCard: a Gigabit IPS on the Network Card. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[36] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. MemScale: Active Low-Power Modes for Main Memory. In *ACM Intrnational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[37] L. Deri. Improving Passive Packet Capture: Beyond Device Polling. In *International System Administration and Network Engineering Conference (SANE)*, 2004.

[38] L. Deri. High-Speed Dynamic Packet Filtering. *Journal of Network and Systems Management*, 15(3), 2007.

[39] L. Deri, J. Gasparakis, P. Waskiewicz, and F. Fusco. Wire-Speed Hardware-Assisted Traffic Filtering with Mainstream Network Adapters. *Advances in Network-Embedded Management and Applications*, 2011.

[40] P. J. Desnoyers and P. Shenoy. Hyperion: High Volume Stream Archival for Retrospective Querying. In *USENIX Annual Technical Conference (ATC)*, 2007.

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

[42] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[43] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational Experiences with High-Volume Network Intrusion Detection. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[44] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the Resource Consumption of Network Intrusion Detection Systems. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[45] J. Drobisz and K. J. Christensen. Adaptive Sampling Methods to Determine Network Traffic Statistics including the Hurst Parameter. In *Annual IEEE Conference on Local Computer Networks (LCN)*, 1998.

[46] N. Duffield. Sampling for Passive Internet Measurement: A Review. *Statistical Science*, 19(3), 2004.

[47] N. Duffield, C. Lund, and M. Thorup. Charging from Sampled Network Usage. In *ACM SIGCOMM Workshop on Internet Measurement*, 2001.

[48] N. Duffield, C. Lund, and M. Thorup. Properties and Prediction of Flow Statistics from Sampled Packet Streams. In *ACM SIGCOMM Workshop on Internet measurment*, 2002.

[49] N. Duffield, C. Lund, and M. Thorup. Flow Sampling Under Hard Resource Constraints. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2004.

[50] N. G. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. In *ACM SIGCOMM Conference on Data Communication*, 2000.

[51] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a Better NetFlow. In *ACM SIGCOMM Conference on Data Communication*, 2004.

[52] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transanctions on Computer Systems*, 21(3), 2003.

[53] W. Fang and L. Peterson. Inter-AS Traffic Patterns and Their Implications. In *Global Telecommunications Conference*, 1999.

[54] Y. Fu, L. Cherkasova, W. Tang, and A. Vahdat. EtE: Passive End-to-End Internet Service Performance Monitoring. In *USENIX Annual Technical Conference (ATC)*, 2002.

[55] F. Fusco and L. Deri. High Speed Network Traffic Analysis with Commodity Multi-core Systems. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2010.

[56] F. Fusco, X. Dimitropoulos, M. Vlachos, and L. Deri. pcapIndex: An Index for Network Packet Traces with Legacy Compatibility. *SIGCOMM Computer Communication Review (CCR)*, 42(1), 2012.

[57] F. Fusco, M. Vlachos, and X. Dimitropoulos. RasterZip: Compressing Network Monitoring Data with Support for Partial Decompression. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2012.

[58] J. García-Dorado, F. Mata, J. Ramos, P. Santiago del Río, V. Moreno, and J. Aracil. High-Performance Network Traffic Processing Systems Using Commodity Hardware. In *Data Traffic Monitoring and Analysis*, volume 7754. 2013.

[59] J. M. González and V. Paxson. Enhancing Network Intrusion Detection with Integrated Sampling and Filtering. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[60] M. Grossglauser and J. Rexford. Passive Traffic Measurement for IP Operations. In *The Internet as a Large-Scale Complex System*. 2005.

[61] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *USENIX Security Symposium*, 2007.

[62] D. Gugelmann, D. Schatzmann, and V. Lenders. Horizon Extender: Long-term Preservation of Data Leakage Evidence in Web Traffic. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[63] M. Gupta and S. Singh. Greening of the Internet. In *ACM SIGCOMM Conference on Data Communication*, 2003.

[64] P. Gupta and N. McKeown. Algorithms for Packet Classification. *IEEE Network Special Issue*, 15(2), 2001.

[65] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *ACM SIGCOMM Conference on Data Communication*, 2010.

[66] M. Handley, V. Paxson, and C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *USENIX Security Symposium*, 2001.

[67] P. Hazel. PCRE: Perl Compatible Regular Expressions. `http://www.pcre.org`.

[68] E. A. Hernandez, M. C. Chidester, and A. D. George. Adaptive Sampling for Network Management. *Journal of Network and Systems Management*, 9(4), 2001.

[69] N. Hohn and D. Veitch. Inverting sampled traffic. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2003.

[70] G. Iannaccone, C. Diot, D. McAuley, A. Moore, I. Pratt, and L. Rizzo. The CoMo White Paper, 2004. `http://como.intel-research.net/pubs/como.whitepaper.pdf`.

[71] IETF. Packet sampling working group cgarter. `http://www.ietf.org/html.charters/psamp-charter.html`.

[72] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese. Network Monitoring Using Traffic Dispersion Graphs (TDGs). In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2007.

[73] C. M. Inacio and B. Trammell. YAF: Yet Another Flowmeter. In *USENIX Large Installation System Administration Conference (LISA)*, 2010.

[74] Intel. 82599 10 GbE Controller Datasheet. `http://download.intel.com/design/network/datashts/82599_datasheet.pdf`.

[75] Intel Server Adapters. Receive Side Scaling on Intel Network Adapters. `http://www.intel.com/support/network/adapter/pro100/sb/cs-027574.htm`.

[76] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: Packet Filtering for Low-Cost Network Monitoring. In *IEEE Workshop on High-Performance Switching and Routing (HPSR)*, 2002.

[77] H. Jiang and C. Dovrolis. Passive Estimation of TCP Round-Trip Times. *SIGCOMM Computer Communication Review (CCR)*, 32(3), 2002.

[78] S. Kandula, R. Chandra, and D. Katabi. What's going on?: Learning Communication Rules in Edge Networks. In *ACM SIGCOMM Conference on Data Communication*, 2008.

[79] T. Karagiannis, A. Broido, M. Faloutsos, and K. claffy. Transport Layer Identification of P2P Traffic. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2004.

[80] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: multilevel traffic classification in the dark. In *ACM SIGCOMM Conference on Data Communication*, 2005.

[81] S. Khan and I. Traore. A Prevention Model for Algorithmic Complexity Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2005.

[82] J. Kirrage, A. Rathnayake, and H. Thielecke. Static Analysis for Regular Expression Denial-of-Service Attacks. 7873, 2013.

[83] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer. Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2005.

[84] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *IEEE Symposium on Security and Privacy*, 2002.

[85] W. Lee, J. B. D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang. Performance Adaptation in Real-Time Intrusion Detection Systems. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

[86] T. Limmer and F. Dressler. Improving the Performance of Intrusion Detection using Dialog-based Payload Aggregation. In *IEEE Global Internet Symposium (GI)*, 2011.

[87] Y.-D. Lin, P.-C. Lin, T.-H. Cheng, I.-W. Chen, and Y.-C. Lai. Low-Storage Capture and Loss Recovery Selective Replay of Real Flows. *IEEE Communications Magazine*, 50(4), 2012.

[88] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang. Is Sampled Data Sufficient for Anomaly Detection? In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2006.

[89] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching Network Security Analysis with Time Travel. In *ACM SIGCOMM Conference on Data Communication*, 2008.

[90] E. P. Markatos, D. N. Pnevmatikatos, M. D. Flouris, and M. G. H. Katevenis. Web-Conscious Storage Management for Web Proxies. *IEEE/ACM Transactions on Networking*, 10(6), 2002.

[91] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Winter USENIX Conference*, 1993.

[92] S. McCanne, C. Leres, and V. Jacobson. Libpcap. `http://www.tcpdump.org/`. Lawrence Berkeley Laboratory.

[93] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic Matrix Estimation: Existing Techniques and New Directions. In *ACM SIGCOMM Conference on Data Communication*, 2002.

[94] J. Menon. Storage Futures and Research. `http://indico.cern.ch/getFile.py/access?contribId=%20526&sessionId=21&resId=0&materialId=slides&confId=0`.

[95] L. Michael and G. Lior. The Effect of Packet Reordering in a Backbone Link on Application Throughput. *IEEE Network*, 16(5), 2002.

[96] J. Micheel, H.-W. Braun, and I. Graham. Storage and bandwidth requirements for passive internet header traces. In *Proceedings of the Workshop on Network-Related Data Management*, 2001.

[97] A. G. Miklas, S. Saroiu, A. Wolman, and A. D. Brown. Bunker: A Privacy-Oriented Platform for Network Tracing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[98] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3), 1997.

[99] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto. Identifying Elephant Flows Through Periodically Sampled Packets. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2004.

[100] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2003.

[101] Myricom. 10G-PCIE-8B-S Single-Port 10-Gigabit Ethernet Network Adapters. `https://www.myricom.com/images/stories/10G-PCIE-8B-S.pdf`.

[102] Myricom. Myricom Sniffer10G. `http://www.myricom.com/scs/SNF/doc/`, 2010.

[103] J. Novak and S. Sturges. Target-Based TCP Stream Reassembly. `http://assets.sourcefire.com/snort/developmentpapers/stream5-model-Aug032007.pdf`, 2007.

[104] T. Oetiker. Rrdtool. `http://oss.oetiker.ch/rrdtool/`.

[105] A. Papadogiannakis, D. Antoniades, M. Polychronakis, and E. P. Markatos. Improving the Performance of Passive Network Monitoring Applications Using Locality Buffering. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2007.

[106] A. Papadogiannakis, A. Kapravelos, M. Polychronakis, E. P. Markatos, and A. Ciuffoletti. Passive End-to-end Packet Loss Estimation for Grid Traffic Monitoring. In *CoreGRID Integration Workshop*, 2006.

[107] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos. Improving the Accuracy of Network Intrusion Detection Systems Under Load Using Selective Packet Discarding. In *ACM European Workshop on System Security (EUROSEC)*, 2010.

[108] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos. RRDtrace: Long-term Raw Network Traffic Recording Using Fixed-size Storage. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010.

[109] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos. Tolerating Overload Attacks Against Packet Capturing Systems. In *USENIX Annual Technical Conference (ATC)*, 2012.

[110] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos. Scap: Stream-Oriented Network Traffic Capture and Analysis for High-Speed Networks. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2013.

[111] A. Papadogiannakis, G. Vasiliadis, D. Antoniades, M. Polychronakis, and E. P. Markatos. Improving the Performance of Passive Network Monitoring Applications with Memory Locality Enhancements. *Computer Communications*, 35(1), 2012.

[112] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine Grained Energy Accounting on Smartphones with Eprof. In *ACM European Conference on Computer Systems (EuroSys)*, 2012.

[113] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24), 1999.

[114] V. Paxson, R. Sommer, and N. Weaver. An Architecture for Exploiting Multi-Core Processors to Parallelize Network Intrusion Prevention. In *IEEE Sarnoff Symposium*, 2007.

[115] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *ACM European Conference on Computer Systems (EuroSys)*, 2012.

[116] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level Polymorphic Shellcode Detection using Emulation. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2006.

[117] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. An Empirical Study of Real-world Polymorphic Code Injection Attacks. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[118] M. Ponec, P. Giura, H. Brönnimann, and J. Wein. Highly efficient techniques for network forensics. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[119] T. H. Ptacek and T. N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., 1998.

[120] F. Raspall and S. Sallent. Adaptive Shared-State Sampling. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2008.

[121] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling Real-Time Querying of Live and Historical Stream Data. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, 2007.

[122] B. Ribeiro, D. Towsley, T. Ye, and J. C. Bolot. Fisher Information of Sampled Packets: An Application to Flow Size Estimation. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2006.

[123] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference (ATC)*, 2012.

[124] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *USENIX Large Installation System Administration Conference (LISA)*, 1999.

[125] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *Annual Linux Showcase & Conference (ALS)*, 2001.

[126] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *ACM SIGCOMM Computer Communication Review*, 29(5), 1999.

[127] L. Schaelicke, T. Slabach, B. J. Moore, and C. Freeland. Characterizing the Performance of Network Intrusion Detection Sensors. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2003.

[128] L. Schaelicke, K. Wheeler, and C. Freeland. SPANIDS: a Scalable Network Intrusion Detection Loadbalancer. In *Conference on Computing Frontiers (CF)*, 2005.

[129] F. Schneider and J. Wallerich. Performance Evaluation of Packet Capturing Systems for High-Speed Networks. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2005.

[130] F. Schneider, J. Wallerich, and A. Feldmann. Packet Capture in 10-gigabit Ethernet Environments Using Contemporary Commodity Hardware. In *International Conference on Passive and Active Network Measurement (PAM)*, 2007.

[131] U. Shankar and V. Paxson. Active Mapping: Resisting NIDS Evasion without Altering Traffic. In *IEEE Symposium on Security and Privacy*, 2003.

[132] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power Containers: an OS Facility for Fine-grained Power and Energy Management on Multicore Servers. In *ACM Intrnational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[133] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[134] SMC Networks. SMC10GPCIe-XFP Tiger Card 10G PCIe 10GbE XFP Server Adapter. `http://www.smc.com/files/AF/ds_SMC10GPCIe_XFP.pdf`.

[135] R. Smith, C. Estan, and S. Jha. Backtracking Algorithmic Complexity Attacks Against a NIDS. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[136] R. Smith, C. Estan, and S. Jha. XFA: Faster Signature Matching with Extended Automata. In *IEEE Symposium on Security and Privacy*, 2008.

[137] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In *ACM SIGCOMM Conference on Data Communication*, 2008.

[138] Solarfrare. 10GbE LOM/Controller Family. `http://www.solarflare.com/Content/UserFiles/Documents/Solarflare_SFC9000_10GbE_Controller_Brief.pdf`.

[139] R. Sommer, V. Paxson, and N. Weaver. An Architecture for Exploiting Multi-Core Processors to Parallelize Network Intrusion Prevention. *Concurrency and Computation: Practice and Experience*, 21(10), 2009.

[140] N. T. Spring and D. Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *ACM SIGCOMM Conference on Data Communication*, 2000.

[141] T. Taylor, S. E. Coull, F. Monrose, and J. McHugh. Toward Efficient Querying of Compressed Network Payloads. In *USENIX Annual Technical Conference (ATC)*, 2012.

[142] The Tcpdump Group. tcpdump. `http://www.tcpdump.org/`.

[143] P. Trimintzios, M. Polychronakis, A. Papadogiannakis, M. Foukarakis, E. P. Markatos, and A. Øslebø. DiMAPI: An Application Programming Interface for Distributed Network Monitoring. In *IFIP/IEEE Network Operations and Management Symposium (NOMS)*, 2006.

[144] P. Tune and D. Veitch. Towards Optimal Sampling for Flow Size Estimation. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2008.

[145] A. Turner. Tcpreplay. `http://tcpreplay.synfin.net/trac/`.

[146] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[147] J. Van Der Merwe, R. Caceres, Y. Chu, and C. Sreenan. mmdump: A Tool for Monitoring Internet Multimedia Traffic. *ACM SIGCOMM CCR*, 30(5), 2000.

[148] G. Varghese, J. A. Fingerhut, and F. Bonomi. Detecting Evasion Attacks at High Speeds without Reassembly. In *ACM SIGCOMM Conference on Data Communication*, 2006.

[149] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[150] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and Robust TCP Stream Normalization. In *IEEE Symposium on Security and Privacy*, 2008.

[151] S. Woo and K. Park. Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. Technical report, Technical report, KAIST, 2012.

[152] Z. Wu, M. Xie, and H. Wang. Swift: A Fast Dynamic Packet Filter. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[153] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, and E. P. Markatos. An Active Splitter Architecture for Intrusion Detection and Prevention. *IEEE Transactions on Dependable and Secure Computing*, 03(1), 2006.

[154] L. Yang, R. Karim, V. Ganapathy, and R. Smith. Improving NFA-Based Signature Matching Using Ordered Binary Decision Diagrams. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2010.

[155] V. Yegneswaran, P. Barford, and J. Ullrich. Internet Intrusions: Global Characteristics and Prevalence. 2003.

[156] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. In *IEEE International Conference on Network Protocols (ICNP)*, 2004.

[157] S. Yusuf and W. Luk. Bitwise Optimised CAM for Network Intrusion Detection Systems. In *International Conference on Field Programmable Logic and Applications*, 2005.