Network Architectures
and Services
NET 2014-05-2

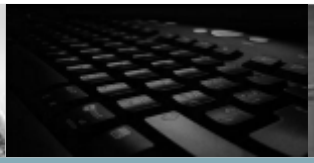
**Dissertation**

# Traffic Analysis on High-Speed Internet Links

Lothar Braun

Technische Universität München





TECHNISCHE UNIVERSITÄT MÜNCHEN
Institut für Informatik
Lehrstuhl für Netzarchitekturen und Netzdienste

# Traffic Analysis on High-Speed Internet Links

Lothar Braun

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

| | | |
|---|---|---|
| Vorsitzender: | | Univ.-Prof. Dr. Martin Bichler |
| Prüfer der Dissertation: | 1. | Univ.-Prof. Dr.-Ing. Georg Carle |
| | 2. | Univ.-Prof. Dr.-Ing. Tanja Zseby, Technische Universität Wien, Österreich |

Die Dissertation wurde am 12.12.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.04.2014 angenommen.

Cataloging-in-Publication Data
Lothar Braun
*Traffic Analysis on High-Speed Internet Links*
Dissertation, May 2014
Network Architectures and Services, Department of Computer Science
Technische Universität München

ISBN 3-937201-42-4
ISSN 1868-2634 (print)
ISSN 1868-2642 (electronic)
Network Architectures and Services NET-2014-05-2
Series Editor: Georg Carle, Technische Universität München, Germany

**Abstract**

The past years have seen an increase in the importance of computer networks for many tasks in day-to-day life. Network services are crucial for many business work-flows and become more important for the private life driven by new services such as social networks or online video streaming portals. As the need for network service availability increases, operators see a growing need for understanding the current state of their networks. Monitoring techniques for detecting network failures, attacks on end systems, or potential bottlenecks that could be mitigated by careful network optimization receive more attention in the research and business community.

Many current traffic analysis systems employ deep packet inspection (DPI) in order to analyze network traffic. These systems include intrusion detection systems, software for network traffic accounting, traffic classification, or systems for monitoring service-level agreements. Traffic volumes and link speeds of current enterprise and ISP networks, however, transform the process of inspecting traffic payload into a challenging task.

A traffic analysis setup needs to be properly configured in order to meet the challenges posed by traffic volumes in current high-speed networks. This dissertation evaluates the performance of current packet capturing solutions of standard operating systems on commodity hardware. We identify and explain bottlenecks and pitfalls within the capturing stacks, and provide guidelines for users on how to configure their capturing systems for optimal performance. Furthermore, we propose improvements to the operating system's capturing processes that reduce packet loss, and evaluate their impact on capturing performance.

Depending on the computational complexity of the desired traffic analysis application, even the best-tuned capturing setups can suffer packet loss if the employed hardware is short in available computational resources. We address this problem by presenting and evaluating new sampling algorithms that can be deployed in front of a traffic analysis application to reduce the amount of inspected packets without degrading the results of the analysis significantly. These algorithms can be used in conjunction with multicore-aware network traffic analysis setups for exploiting the capabilities of multi-core hardware. The presented analysis architecture is demonstrated to be suitable for live traffic measurements for security monitoring, for the analysis of security protocols and for traffic analysis for network optimization.

# Acknowledgements

This thesis would have not been possible without the support of many people. I would like to thank my colleagues at the chair for Network Architectures and Services, and the many good students who contributed to this work. The fun and productive work environment that was created by all of them allowed me to collaborate with different people on different projects. In particular, I would like to thank Gerhard Münz, who advised me when I was a student and in the early years of my PhD project. I learned a lot from his guidance. Much of this work has been previously published as papers at conferences and workshops, and I would like to thank all of my co-authors for their contributions. Thank you Cornelius Diekmann, Oliver Gasser, Ralph Holz, Holger Kinkelin and Johann Schlamp, for reading and correcting this document and for giving me valuable feedback.

I would like to thank Prof. Dr.-Ing. Georg Carle for supervising this thesis, and for giving me the opportunity to work on many interesting projects in his group. My thanks also go to Prof. Dr.-Ing. Tanja Zseby for being my second supervisor, and to Prof. Dr. Martin Bichler for the organization of the examination procedure.

I am also very thankful to my parents Elvira and Alexander Braun who have always supported and believed in me. Without them, I would never have been in the position to start this PhD project. Finally, I would like to thank my wife Maja for her support, understanding and patience during all these years. She has been there for me throughout the whole time of this long project.

**Previously published parts of this thesis:**


**[1]** Lothar Braun, Gerhard Münz, and Georg Carle, "Packet Sampling for Worm and Botnet Detection in TCP Connections," in 12th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2010), Osaka, Japan, Apr. 2010.

**[2]** Lothar Braun, Alexander Didebulidze, Nils Kammenhuber, and Georg Carle, "Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware," in Proceedings of the 10th Annual Conference on Internet Measurement (IMC 2010), Melbourne, Australia, Nov. 2010.

**[3]** Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. "The SSL Landscape - A Thorough Analysis of the X.509 PKI Using Active and Passive Measurements" in Proceedings of the 11th Annual Internet Measurement Conference (IMC 2011), Berlin, Germany, November 2011.

**[4]** Lothar Braun, Alexander Klein, Georg Carle, Helmut Reiser, and Jochen Eisl. "Analyzing Caching Benefits for YouTube Traffic in Edge Networks - A Measurement-Based Evaluation," in Proceedings of the 13th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2012), Maui, Hawaii, April 2012.

**[5]** Lothar Braun, Mario Volke, Johann Schlamp, Alexander von Bodisco, and Georg Carle. "Flow-Inspector: A Framework for Visualizing Network Flow Data using Current Web Technologies," in First IMC Workshop on Internet Visualization (WIV 2012), Boston, MA, November 2012.

**[6]** Lothar Braun, Cornelius Diekmann, Nils Kammenhuber, Georg Carle, "Adaptive Load-Aware Sampling for Network Monitoring on Multicore Commodity Hardware," in Proceedings of the IEEE/IFIP Networking 2013, New York, NY, May 2013.

# Contents

# Part I

# Introduction and Network Monitoring Fundamentals

# 1 Introduction

## 1.1 Motivation for Traffic Analysis and Network Monitoring

Many applications on today's computers rely on network connectivity for various purposes. These include business applications, which use network services to a growing extent. Microsoft for example started adding online services with its Microsoft Office 365 product line in 2011 [7]. Other applications, such as Google Docs, are completely hosted in a data center environment and accessible as a Web service.

Network problems can have severe negative influence on application performance and user experience. Connectivity problems can result in users not being able to access or use their desired services or applications. Due to the increasing importance of the network, operators of company networks and Internet Service Providers (ISPs) are interested in providing good services and detecting possible problems as fast as possible. Several services such as Voice Over IP, video streaming or online gaming do not only require network connectivity but have special requirements concerning network performance. Such services create pressure on operators to evaluate and improve the performance of their networks.

A recent study by the Lawrence Berkeley Labs estimates that large portions of the U.S. work force rely server-based production software to some extent [8]. Network services are not only important for business applications, but are important for the private consumer market as well. A study by Cisco estimates that the share of the consumer traffic segment exceeds the traffic produced by government or business users in Wide Area Networks [9]. These studies emphasize the importance of network reliability both for company network operators as well as ISPs for the consumer market.

A prerequisite for network optimization and problem detection is a good understanding of the state of the network and the traffic that is transported in it. Operators are therefore interested in the use of network monitoring to gain insight into various aspects of the network. Network monitoring is a complex task that can make use of many different techniques and tools. The choice of techniques and tools always depends on the desired analysis goal. Important goals of network monitoring are the detection of security risks or breaches or the identification of bottlenecks for network optimization. In the following, we will discuss these goals in more detail.

The identification of security problems is a particular important task for network monitoring systems. Computers that are infected with malware and participate in a botnet can be used to stage Denial-of-Service (DoS) attacks against networks or end hosts. The scale of the problem has motivated researchers to present numerous approaches to detect and mitigate DoS attacks [10], and companies such as Arbor Networks [11] successfully sell products that help operators to protect their networks from these types of attacks. Consequent *detection* and removal of *worms and bot clients* can help to mitigate the threat of DoS attacks by removing the sources of the attack traffic.

Reactive detection of infected machines is only one way to employ monitoring to enhance the security of operator networks. Proactive network analysis can help to identify potential problems before they are exploited by adversaries. Security scanners such as NMAP [12] or Nessus [13] can help to detect vulnerable services, and can help to raise the administrator's awareness of vulnerabilities. Vulnerabilities can be unpatched software with known security holes or problems with the deployment of services and infrastructures. An example for such deployment problems can be found in one of the Internet's most important security protocols: TLS. TLS and its predecessor SSL are used to protect the communication of large numbers of Internet protocols, providing data confidentiality, integrity and authenticity. A central component of the authentication and key distribution process that is crucial for the deployment of TLS/SSL is the Public Key Infrastructure (PKI). It is well known that the practical implementation of a PKI is difficult and can lead to potential problems if not done right [14]. *Deployment Analysis* with network monitoring techniques can help operators and researchers to assess the quality of the deployed security infrastructure.

Another important field of activity for network monitoring is the identification of potential for *network optimization*. The requirements that originate from the increased usage of network connectivity and bandwidth in today's applications put pressure on operators to provide optimal use of the available resources and to optimize the performance of their networks. One of the major drivers of demand for bandwidth is video traffic [9]. Operators therefore think about the deployment of in-network caches to reduce the bandwidth of traffic that is needed to download the video content. The benefits and costs of a cache largely depend on the users content access behavior. Network monitoring techniques can help to provide an understanding of this access behavior in a certain network. This knowledge can be used to assess the benefits of the deployment of such a cache.

The diversity of the previously discussed applications of network monitoring calls for highly flexible tools that allow the investigation of many different aspects of computer networks and the traffic that is transported in them. If the monitoring process includes traffic measurements, which analyze the traffic in the network, then it has to cope with the ever increasing amount of bandwidth in current high-speed networks. Processing large amounts of traffic on very fast links at line-speed can be a difficult task, especially if the performed analysis is complex. Special-purpose monitoring hardware based on Field Programmable Gate Arrays (FPGAs) has been a proposed solution for high-speed networks, e.g. [15, 16, 17, 18]. However, a problem with this special purpose hardware

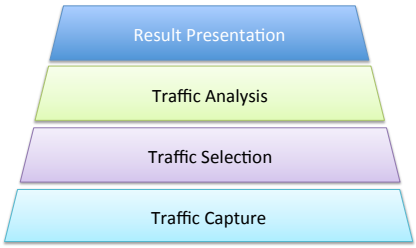



**Figure 1.1:** Building blocks of a traffic analysis system.

is that it requires sophisticated programming, which limits flexibility in deploying diverse forms of analysis.

With the increased capabilities of modern off-the-shelf commodity hardware, traffic analysis with multi-core standard hardware is seen to have the potential of providing enough resources to be an alternative to FPGA, even in high-speed networks [19]. However, the use of commodity hardware requires careful tuning and optimization of the analysis system in order to yield good performance. Consideration of overload scenarios is also important in case the complexity of the analysis exceeds the available resources. In general, any traffic analysis system has to perform multiple tasks, beginning with the acquisition of traffic and ending with the presentation of the analysis results to a user. Figure 1.1 shows the building blocks that compose a traffic analysis system.

This thesis provides contributions to the lower three building blocks:

- We evaluate and improve the performance of packet capture stacks of general purpose operating systems for traffic analysis on commodity hardware in high-speed networks.
- We provide sampling algorithms that complement the existing body of work on sampling with traffic selection that provides the necessary input to analyses like the ones described in this section.
- We demonstrate the suitability of our approach for
  - *worm and botnet* detection
  - *deployment analysis of the TLS/SSL PKI infrastructure*
  - traffic analysis to estimate the benefits of *caching of user generated video traffic* within end-networks.

The following section will discuss the problems that this thesis wants to solve and presents the contributions of the thesis.

## 1.2 Problem Statement and Contributions

The problems in this thesis can be grouped into two categories. First, we work on problems that are not specific to certain analyses but traffic analysis systems in general. These problems are related to the research problems of how to build a efficient traffic analysis system and how to cope with overload scenarios.

The second group of problems is related to specific analyses. In this thesis, we want to perform a number of analyses from the fields security monitoring, deployment analysis and traffic analysis for network optimization. Hence, the problems in this group focus on how to employ traffic analysis for specific tasks.

However, it is not possible to separate clearly between the individual problems and questions: Sampling is a technique that can be used to cope with overload scenarios. The applicability of a particular sampling algorithm, however, is highly depended on the analysis task at hand. In the following, we will discuss the research problems from both groups.

### 1.2.1 Traffic Capture

Traffic analysis on commodity hardware provides benefits compared to the application of special purpose hardware built on FPGA. It allows for the simple creation of flexible monitoring software and allows operators to easily switch to new software if the goals of traffic analysis evolve. New feature developments or software upgrades can be more easily incorporated into existing monitoring setups.

Regardless of the specific monitoring application that is in use in a particular environment, the analysis system has to deal with the ever increasing amount of traffic in today's networks. The available bandwidth on modern high-speed links can result in excessive load on the analysis system, overwhelming the available resources. Such excessive load can result in packet loss on the device, which in turn can lead to problems in the analysis. Research shows that packet loss due to system overload can have serious negative impact on the results of an analysis process.

Commodity hardware is often managed by general-purpose operating systems, which are developed with general purpose computing in mind. The software stack is therefore not optimized for traffic analysis, and the packet capture mechanisms employed in these systems often struggle on high-speed networks [20]. Furthermore, different operating systems provide different approaches for packet capture. Researchers also propose new improvements and enhancements that promise even better performance. However, to the users it is often unclear which of the approaches yields best performance, and how to configure the systems for their analysis setup.

Identifying the best approach and the best implementation for packet capture is difficult. Modern operating systems and their network stacks evolve over time, and new approaches to packet capture

are designed. Performance evaluations in test setups can help to answer the question for the best-performing approach. However, a number of different tests can be conducted to show the performance of a system.

In this thesis, we work on the problem of how to perform benchmarks in a way that allows thorough analysis of different approaches. Our goal is to identify how we need to define tests that provide insight into the performance of an approach under various circumstances.

**Contribution:**

We identify and discuss different approaches for packet capture provided by the network stacks of different operating systems. This includes several improvements to these standard stacks that were presented by researchers. We build an evaluation and test setup that allows us to assess the performance of the different capture stacks and the proposed improvements. We show that simple tests, as performed in many related work, are not sufficient to provide thorough analysis of capture approaches.

Our analysis reveals that a good testing methodology can yield in deeper insight into the performance under various conditions. Even more surprising, we show with our testing methodology that an approach that has superior performance in one type of traffic analysis can have substantial packet loss for other analyses.

Based on our evaluation, we identify bottlenecks and develop an enhancement that improves the performance of the capture system in certain scenarios. We provide guidelines for users and developers on how to tune their packet capture systems.

### 1.2.2 Traffic Selection

While the performance of commodity hardware has improved significantly, and is expected to improve further, some deployments of commodity hardware-based traffic analysis setups can still suffer from packet loss. Traffic analysis is often performed online, and processing times for incoming packets must match packet inter-arrival times on average. Otherwise, system buffers will fill up to the point where packets must be discarded.

Packet loss can be substantial, as shown in Figure 1.2. The figure displays a case study of a real security monitoring setup that analyzed traffic on a 10GE link of a large university network. The system was deployed to monitor the border gateway links between the university network and the Internet. It performed security analysis using the Intrusion Detection System Snort [21], with a botnet-specific rule set. The difference between the incoming packet rate and the number of analyzed packets is evident. This particular setup observed packet loss rates of more than 50% during the peak traffic times, due to the high complexity of the analysis.

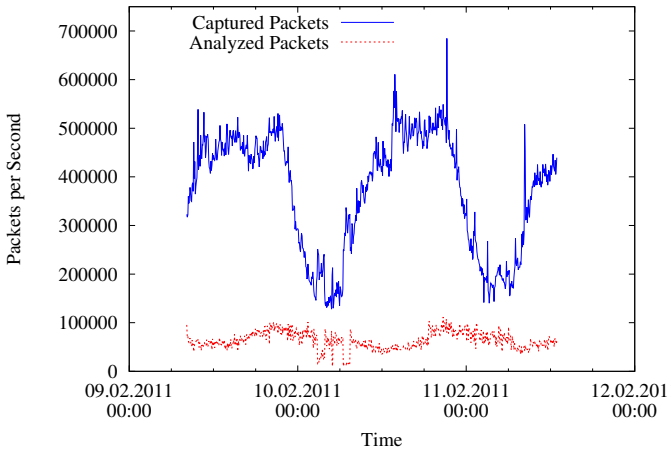



**Figure 1.2:** Case study: Packet loss in high speed networks.

Coping with overload scenarios is important for many analysis algorithms, and has therefore been an active research topic. If it is not possible to improve the performance of the analysis, then the system needs to make choice on which parts of the traffic to analyze. Overloaded packet analysis systems have been a problem for a long time, and triggered a large body of research for sampling techniques.

Sampling a *good* subset of packets from the available traffic can be a difficult task. For example in the field of security monitoring: Whenever a sampling algorithm drops malicious packets, an attack or system compromise may go unnoticed, as the intrusion detection system does not analyze the non-sampled traffic. A *good* subset of the packets includes packets that carry malicious payload, or packets that are necessary for the detection engine. Previous research shows that sampling can distort anomaly detection metrics [22], or can have severe negative impact on the detection rates of some analysis algorithms [23, 24]. Choosing an appropriate sampling algorithm that provides a good share of the overall traffic for such algorithms is important for the use in scenarios where the computational resources are not sufficient.

We introduced a number of important traffic analysis applications in Section 1.1. The problem that we want to solve in this thesis is to enable these and similar analyses in traffic analysis setups that are too short in resources to consume all the traffic in the network. In order to achieve this goal, we propose the use of a certain sampling mechanism, and propose algorithms that implement this mechanism.

**Contribution:**

In this thesis, we work on sampling algorithms that provide good sampling results for the traffic analysis process that are in scope of this work[1]. The sampling focuses on payload from the beginning of TCP connections or UDP bi-flows [25]. We discuss why we think this sampling is

1 Throughout this thesis we will discuss additional application areas that can benefit from this type of sampling.

beneficial for a number of traffic analysis applications. Furthermore, we present two algorithms that implement the proposed sampling, and evaluate their applicability in high-speed networks.

The first algorithm makes use of the data structure Bloom filters [26] to sample the first *N* bytes of payload from TCP connections. We develop an algorithm that implements a simplified TCP connection tracking and makes use of the filters to store the required connection states. The use of Bloom filters allows us to build a state tracker with constant memory requirements and constant computational complexity for state lookup that is independent from the observed connection characteristics. We assess the errors that are introduced due to the use of Bloom filters and show that they can be configured to provide accurate sampling results under normal conditions and acceptable errors under extreme conditions with an unexpectedly high number of connections.

Our second algorithm provides sampling of the beginning of bi-flows [25] using conventional state tracking that allows to sample from TCP connections and UDP bi-flows. The algorithm belongs to the class of adaptive algorithms and chooses the number of bytes to sample based on the packet rate on the network and the processing capabilities of the traffic analysis process. We evaluate the algorithm in real traffic analysis setups on high-speed links and show its applicability under these conditions.

### 1.2.3 Traffic Analysis

Section 1.1 motivated several important traffic analyses, which will be performed throughout this thesis. In the following, we will discuss the problems and contributions of this thesis with respect to the particular analysis tasks.

#### 1.2.3.1 Worm and Botnet Detection

The detection of worms and botnets has been considered an important topic for a number of years. Many researchers presented numerous approaches that aim at the detection of traffic that is related to computers that are infected with malware. A major concern for high-speed networks is the complexity of the proposed analysis.

Especially systems that employ Deep Packet Inspection (DPI) can have the potential of requiring lots of computational resources, which can lead to resource exhaustion on the traffic analysis system. This can be a major obstacle for the use of these analysis techniques in high-speed networks. We want to enable the use of resource intensive security monitoring techniques in high-speed networks in constraint resource environments by the use of sampling.

As part of the thesis, we present sampling algorithms that aim at providing good analysis results for worm and botnet detection. We therefore focus on the problem of determining whether our proposed sampling is suitable for this analysis task.

**Contribution:**

We evaluate the impact of our proposed sampling of beginning of TCP connections and UDP bi-flows to DPI-based analysis of network traffic for the detection of worm and botnet traffic. In experiments with artificially generated malware traffic and live traffic measurements in a large university network environment, we determine the impact of our sampling algorithms on the detection rate of signature-based Intrusion Detection Systems (IDS). Furthermore, we examine how more sophisticated tools for botnet detection, such as BotHunter [27] are influenced by the application of our traffic selection algorithms.

#### 1.2.3.2 Caching benefits for YouTube Traffic

Network optimization is an important task for operators who are under pressure to provide good performance to their users. Traffic measurements can yield important insight into the traffic and can show potential for improvements. A technique that can lead to performance improvements is the use of caching. The benefits of caching highly depend on the way users access content. An important driver for network traffic is video streaming traffic from user-generated content.

Content must be accessed multiple times in order for a cache to provide any benefits. The benefits of a cache therefore depend on the behavior of the users. Traffic measurements can help to determine this user behavior and its implication on caches. Our goal in this thesis is to employ traffic analysis to gain a better understanding of the user behavior for video traffic and its implication for caching benefits.

**Contribution:**

We examine the caching benefits and costs for one of the major players of user-generated video streaming site: YouTube. Using passive traffic measurements over a period of several weeks on a large university network, we are able to record the video download properties of the users in the network. We examine and evaluate properties of the video downloads that are important for a network cache. Furthermore, we employ trace-driven simulation to estimate the costs and benefits that different caching strategies provide on the collected data.

#### 1.2.3.3 Deployment Analysis: The TLS/SSL Public Key Infrastructure

The probably most important security protocols deployed in the Internet are TLS and SSL. A crucial requirement is the necessity to distribute key material and to ensure proper authentication of end points. Both are necessary for the cryptography that is embedded into TLS/SSL to ensure the desired security goals. X.509 Public Key Infrastructure (PKI) is an essential building block that is used to implement the key distribution and authentication of end points. While the mathematic foundation of today's cryptographic protocols are considered solid, doubts about the mechanisms

that are in place to build the PKIs have been voiced. Problems in the PKIs can undermine the security of the complete system. In this thesis, we want to gain a better understanding of the state of the PKI.

We focus on the problem of how to perform measurements that provide insight into the deployment and the use of the infrastructure. Furthermore, we are interested in the analysis results themselves.

**Contribution:**

We provide a large-scale evaluation and assessment of the currently deployed X.509 certificate infrastructure and its use in end networks. We collect certificates by Internet-wide scans of important Web sites over a period of 1.5 years from different vantage points in the world. Using passive traffic measurements, we observe the use of the infrastructure in traffic data from real networks. We collect information on TLS/SSL connections and record all certificates that are exchanged during the connection setup of the monitored TLS/SSL connections.

Based on this data set, we examine the state of the current X.509 deployment in the Internet. We determine properties of the certificates, and, based on these observations, try to gain a better understanding on the underlying processes deployed by the Certification Authorities that issued the certificates.

Our analysis reveals the benefits that the combination of different measurement and traffic analysis techniques can yield compared to an analysis that only evaluates measurements with a single technique.

### 1.2.4 Publications

**Previously published parts of this thesis:**

**[1]** Lothar Braun, Gerhard Münz, and Georg Carle, "Packet Sampling for Worm and Botnet Detection in TCP Connections," in 12th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2010), Osaka, Japan, Apr. 2010.

**[2]** Lothar Braun, Alexander Didebulidze, Nils Kammenhuber, and Georg Carle, "Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware," in Proceedings of the 10th Annual Conference on Internet Measurement (IMC 2010), Melbourne, Australia, Nov. 2010.

**[3]** Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. "The SSL Landscape - A Thorough Analysis of the X.509 PKI Using Active and Passive Measurements" in Proceedings of the 11th Annual Internet Measurement Conference (IMC 2011), Berlin, Germany, November 2011.

**[4]** Lothar Braun, Alexander Klein, Georg Carle, Helmut Reiser, and Jochen Eisl. "Analyzing Caching Benefits for YouTube Traffic in Edge Networks - A Measurement-Based Evaluation," in Proceedings of the 13th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2012), Maui, Hawaii, April 2012.

**[5]** Lothar Braun, Mario Volke, Johann Schlamp, Alexander von Bodisco, and Georg Carle. "Flow-Inspector: A Framework for Visualizing Network Flow Data using Current Web Technologies," in First IMC Workshop on Internet Visualization (WIV 2012), Boston, MA, November 2012.

**[6]** Lothar Braun, Cornelius Diekmann, Nils Kammenhuber, Georg Carle, "Adaptive Load-Aware Sampling for Network Monitoring on Multicore Commodity Hardware," in Proceedings of the IEEE/IFIP Networking 2013, New York, NY, May 2013.

## 1.3 Document Structure

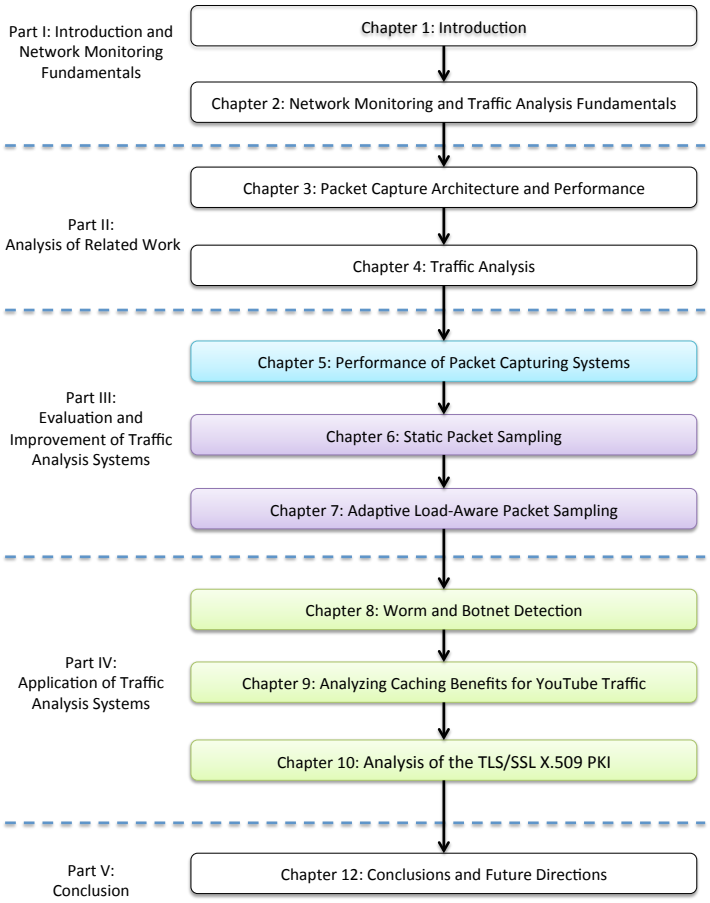



**Figure 1.3:** Chapter guide through this thesis

Figure 1.3 illustrates the structure of this thesis. Its colors map to the building blocks of a traffic analysis system (compare Figure 1.1) and highlights which chapter corresponds to which building block.

This dissertation is divided into five major parts: ”Introduction and Network Monitoring Fundamentals”, ”Analysis of Related Work”, ”Evaluation and Improvement of Traffic Analysis Systems”, ”Application of Traffic Analysis Systems”, and ”Conclusion”. In the remainder of this section, we outline the content of each of the chapters that compose these five parts.

**Chapter 2** introduces and discusses architectures for network monitoring in general. The chapter aims at providing an overview over the techniques and tools that are available to network operators and researchers to gain awareness of the state of their networks. We introduce monitoring and measurement techniques that are used throughout the remainder of the thesis.

Related work that we used to build upon or distinguish ourselves from is introduced and discussed in **Part II** of this thesis. The related work is split into two chapters.

**Chapter 3** focuses on the architecture of packet capture systems in standard operating systems on commodity hardware. It introduces the fundamentally different architecture of two operating systems that are of particular interest in the research community: Linux and FreeBSD. Both operating systems have had improvements to their architecture proposed in research papers. Some of them have been implemented and were eventually included into the operating system main development line. Others are still actively maintained and improved by the researchers who developed the improvements. Due to the importance of good performance of packet capture mechanisms, previous work compared and improved the existing mechanisms. The chapter discusses this previous work in detail in order to derive requirements for our own evaluation setups.

Furthermore, the chapter discusses techniques that must be applied when packet loss occurs: sampling and filter techniques. The chapter introduces the related work on sampling, as well as studies that evaluated the applicability of sampling techniques for certain traffic analysis tasks.

**Chapter 4** discusses work that is related to the analyses in Part IV. This work covers the topics of caching for YouTube video streaming, botnet-related security monitoring and TLS/SSL PKI analysis.

**Part III** contains the contributions of this thesis on packet capture performance studies and improvements. Furthermore, we introduce two novel sampling algorithms that help to cope with high traffic volumes on high-speed networks for various traffic analysis applications.

**Chapter 5** studies the performance of today’s capturing architectures in standard operating systems. We evaluate and compare different capturing solutions for both operating systems, and try to summarize the effects that can lead to bad capturing performance. The chapter aims at future developers and users of capture systems and serves as a resource helping them not to repeat the

pitfalls we and other researchers have encountered. It presents tests and evaluations for thorough testing of traffic analysis setups in different configurations. We identify potential bottlenecks that can lead to performance decreases and thus packet loss. Finally, we propose a modification that can be applied to popular capturing solutions on Linux. It improves capturing performance in a number of scenarios.

**Chapter 6** introduces a sampling algorithm that focuses on sampling the first N bytes of the payload of every observed TCP connection. We develop and implement a sampling algorithm for deployment in high-speed networks with very high packet rates and large numbers of simultaneous TCP connections. The algorithm selects packets containing the first *N* payload bytes of a TCP connection by using a simplified TCP connection tracking mechanism and Bloom filters to store the connection states.

**Chapter 7** complements the work on static sampling of connection beginnings with an algorithm that has a dynamic component. Choosing the right amount of traffic to sample for each connection is a hard task. It is possible for certain tasks to give a good estimate on how many bytes are necessary to get all the desired information for a certain analysis. However, for other tasks, like security monitoring, no good values can be given. An adversary could try to evade sampling by sending *N* bytes of legitimate traffic and start the attack afterwards. The chapter presents an adaptive sampling algorithm that chooses the amount of sampled payload based on the available computing resources of the analysis machine.

**Part IV** focuses on various applications of traffic analysis systems.

**Chapter 8** studies the application of security monitoring for botnet detection. We evaluate the applicability of our sampling approach that samples the first *N* bytes of payload from every connection or bi-flow and scrutinize the detection capabilities of a traffic analysis system that employs our sampling. In order to do this, we use traffic repositories that contain botnet traffic from multiple sources. At first, we analyze traffic traces that contain traffic from dynamic malware analysis systems that run malware samples on analysis systems and allow them to communicate with the open internet. Furthermore, we analyze real-world traffic traces from a large university network.

**Chapter 9** studies the application of traffic analysis for network optimization and cache benefit estimation. Caches in networks can be used in order to reduce the load on the inter-domain link, if they are able to serve content from their internal storage. We study traffic from the YouTube video application in detail and examine its potentials for caching. Several weeks of video download statistics have been collected in order to calculate traffic properties that are relevant for caches. We perform trace-driven simulations of different caching strategies in order to identify the most promising ones.

In **Chapter 10**, we use traffic analysis to estimate the currently deployed security infrastructure in the Internet. As part of our analysis, traffic measurements are used to collect information on the

deployment and use of certificates. We analyze the collected data in order to estimate the state of the currently deployed TLS/SSL security infrastructure.

Finally, **Part V** concludes the thesis by summarizing the findings and results of this work.

# 2 Network Monitoring and Traffic Analysis Fundamentals

Throughout this thesis, we will use the terms *monitoring* and *measurement*, e.g. as in network monitoring or traffic measurement. Monitoring refers to the process of observing the state of the network. Measurement refers to the process of obtaining specific information by the analysis of an object. A traffic measurement can, for example, analyze packets in order to count the number of packets, bytes, and flows in a certain time frame. Hence, traffic measurements are a technique of network monitoring.

Network monitoring aims at the provision of information on the state of a network to administrators or researchers. The acquisition of information is a process that consists of multiple steps. These are often performed by different components within a network monitoring architecture or infrastructure. We identify the following tasks that have to be conducted as part of the monitoring:

- Data acquisition
- Data preparation and normalization
- Data analysis
- Result reporting

The tasks can be performed by different components that form the monitoring architecture. Operators and researchers can use data from network monitoring to gain a better understanding of computer networks. This chapter presents common components and fundamental monitoring techniques. We present traffic measurement techniques that are important throughout this thesis. Furthermore, we discuss other techniques of network monitoring that can be used by researchers to obtain data to validate traffic measurement results, or that can be used to gather information that cannot be obtained by traffic measurements.

## 2.1 Data Sources

### 2.1.1 Log Files

Traffic measurements are the main focus of this thesis. However, in order to perform traffic measurements, a measurement node must be placed at the point of interest. Placement of nodes in arbitrary points of interest is not always possible or feasible. In these cases, researchers can try to make use of data sources that are already available: log files from devices or applications that are already deployed in the network.

Log files can be produced by network components or network server applications. And they can contain valuable information about the device or application integrity, workload or events that are related to the current state of operation. A major benefit of log files is that they can contain information about the state of the device, which can be hard or impossible to determine by measurements. Each device or application should have a an internal state and might be able to provide reasons for failures. The following example can illustrate this fact: Traffic measurements can determine that a host is unreachable, but cannot provide reasons for why the host is not reachable. A switch connected to the host, on the other hand, could see that the physical link to this host is gone. It can therefore provide the additional information on the missing physical link as a reason for the unavailability of the host.

Several researchers successfully used log files to obtain interesting information from network and application usages. Wendell et al. analyzed log files from a Content Distribution Network [28]. They used this data to study the effect of flash crowds and their content request behavior. Potharaju et al. study middleboxes and network failures that are caused by these boxes [29]. Among other sources, they are able to tap into the event logs of the systems, thus receiving information about critical device states and failures that could be detected by the device itself.

### 2.1.2 Configuration Data

Another source of valuable data can help operators to track down problems that might be introduced by a network configuration change. Products such as RANDIC – Really Awesome New Cisco confIg Differ – from Shubbery Networks [30] can help operators to keep their configuration changes in a database for problem tracking and change accountability.

Researchers also showed the usefulness of configuration data for different analyses. Potharaju et al. used configuration data and changes to the configuration in the previously mentioned middlebox study [29]. They obtained configuration files for the middleboxes from a revision control system (RCS). The RCS stored all versions of the configuration files deployed on the middleboxes in conjunction with the time stamps at which changes were applied to the devices' configuration. The information in this repository allowed for a better understanding of reasons for failures: By

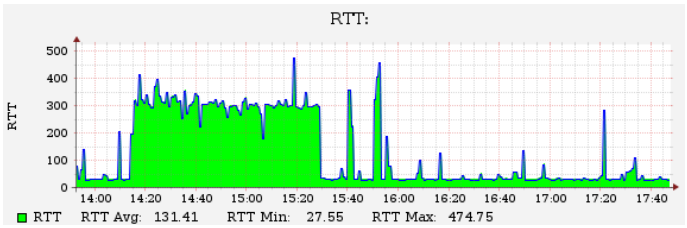



**Figure 2.1:** RTT Measurement

correlating failure events with configuration changes, the authors were able to identify whether failures occurred due to misconfiguration or other types of events.

Configuration data can also be used for other purposes: Benson et al. study data center traffic in several data centers. They used configuration data of network equipment to get information about the data center topology [31].

Kim et al. employ a repository of historic configuration data to infer information about the development and growth of a network [32]. The authors obtained configuration repositories from two university networks that span multiple years. They analyze the repository data to determine the reasons of configuration changes for switches, firewall and routers.

### 2.1.3 Active Traffic Measurements

Another way of gathering the state of the network is to perform active measurements. Active measurements are characterized by the injection of probe packets into the network. Information is derived from the properties of these measurement packets, or from the return traffic that is generated as a result of the probe packets. Examples for active measurements are Round-Trip-Time (RTT) measurements, which can be conducted with ICMP echo packets.

Active measurements can provide valuable information about the state of the network to operators. They play an important role in large-scale enterprise networks that connect multiple sites to headquarters and data centers. RTT measurements can help to constantly monitor the delays on network paths, e.g. on wide area connections between remote offices and the headquarter. Buffer-Bloat in the network is known to be able to increase delays on utilized links [33, 34]. Figure 2.1 shows the RTT over time for a large company network in 2013. The figure plots the RTTs for one site that suffers from these problems due to large file transfers. Active measurements that are performed in regular intervals can detect such problems.

A major benefit of active measurements is the fact that they can be conducted from end systems and do not require changes to the core network. The probes can be performed from any device that is able to inject packets into the network, which can be end hosts or network equipment. Cisco offers a technology called IPSLA [35] that allows making such active measurements from their

switches and routers. Such kind of probing allows end-to-end delay measurements and can be used to determine whether all components in the network work as expected or if problems exist that disturb the correct operation of the network. Bandwidth estimations are another relevant application of active measurements, which can be done by sending pairs or trains of packets [36]. A drawback of active measurements is their intrusiveness: the probing packets can disturb the other traffic on the links that are used for the measurements.

Active measurements can play an important role in network research. Researchers use them in order to gain information about the global structure of the Internet by performing Internet-wide scans [37, 38, 39]. This includes path estimations as well as activities that aim to identify the subnet structure defined by network operators [40]. Active measurements have also been used to study capabilities of home routers [41], for the determination of path characteristics [42, 43], or the comparison of different DNS resolvers [44].

Another important application of active measurements in research is the validation of passive traffic measurement algorithms. Active measurements can be used to inject traffic patterns of known origin and form. A passive analysis can then be performed on the artificially generated measurement packets. The results of the passive measurement can then be compared to the knowledge about the active traffic measurements, by using the active measurements as ground truth. Examples for validation of findings through active measurements can be found in [45]. The authors examined time stamps of NetFlow data from Cisco routers and software probes. They evaluated the accuracy of the time stamps on ground truth that was obtained from injected traffic streams.

### 2.1.4 Passive Traffic Measurements

In contrast to active measurements, passive measurements are non-intrusive and rely on the observation of existing traffic without changing or manipulating its characteristics. Measurement nodes can be deployed everywhere in the network as long as the measurement process has access to the traffic that is to be analyzed. For example, traffic measurements can be performed on any end-system to monitor the traffic at local interfaces. In these scenarios, the passive measurement can only obtain information that is related to the end system, e.g. directed to the system or exchanged on the same broadcast domain.

More information can be retrieved if the passive monitoring system is deployed on central components in the network. The measurements can be configured for different levels of granularity, depending on the type of information that is needed for the analysis. In the following, we use the same terminology as introduced by Münz in [46] for measurement granularity: link-level measurements, packet-level measurements and flow-level measurements.

#### 2.1.4.1 Link-Level Measurements

Link-level measurements can be performed by nearly every network interface and therefore on any device on the network. The simplest link-level measurement counts the number of sent and received packets or bytes. This information is available not only on network components but can be obtained from the operating system of end hosts. Link-level measurements are usually conducted as part of the operation of a device. Updating counters when a packet is sent or received does not put significant additional load on the device. Link-level measurements are a central component in many network monitoring architectures to get an understanding of the amount of traffic that is forwarded by network equipment such as switches or routers. Due to the importance of these counters for network monitoring, several metrics have been standardized by the IETF, e.g. the RMON-MIB [47] and RMON2-MIB [48]. These metrics are obtained and offered by a large number of devices from different vendors. The values stored at the devices can be requested from the device via SNMP [49].

SNMP is considered to be an important tool for network monitoring in company networks. Therefore, large numbers of monitoring tools and platforms with SNMP support exists, both commercial and open source. A comprehensive list can be found at [50].

Researchers also use link-level based measurements for various purposes. Andrey et al. survey 22 research papers that focus on SNMP performance studies up to 2006 [51]. Others use SNMP counters to derive or support the derivation of traffic matrices in networks [52, 53, 54]. Benson et al. use link level measurements in their data center study as well in their effort to understand traffic loads in data centers [31]. The previously discussed middlebox study of Potharaju et al. also made use of interface statistics to assess the impact of failures on traffic volumes [29].

Link-level measurements are the most aggregated form of traffic measurements. They summarize packets from all of the observed traffic streams into a single counter, and therefore remove much information about the communication patterns in the network.

#### 2.1.4.2 Flow-Level Measurements

Flow-level measurements conserve more information about the traffic compared to link-level measurements. Network flows describe the communication streams between end systems or networks. A good definition of flows has been given for IPFIX (IP Flow Information eXport) at the IETF [55]. The definition states that a flow is a unidirectional stream of packets that share a set of common properties, which are also called flow keys. A typical set of flow keys is the so-called IP quintuple that consists of source address, destination address, source port, destination port and the transport protocol.

The basic flow generation is common for all devices that perform flow-level measurements: For every observed packet, a metering process determines the assignment of the packet to a flow by

matching its flow keys. Each flow has several flow properties, such as the observation time of the first and last packet. The flow keys and the other flow properties compose a flow record. Other properties contained in a flow record can be the number of bytes or packets that have been observed for the flow. However, the set of properties is not limited to the aforementioned properties. Modern protocols like IPFIX were designed to be extensible, i.e. to allow the addition of new flow properties. Deri, for example, created a flow probe that is able to embed VoIP quality of service metrics into the flow records [56].

Many flow monitoring devices are designed for generating and exporting flows only. Storage, aggregation or data analysis of network flows is usually offloaded to a separate process on a different machine. Exceptions can be found in software-based monitoring probes such as VERMONT [57], which provide a single application that is capable of the generation, aggregation, and analysis of data.

Many IP-based routers are capable of creating some kind of flow data from the packets that pass their interfaces. Hence, these routers can be configured to create flow data and export it to a central collector. A problem with standard router flow generation can originate from the fact that routers do not generate flows as a main purpose. It is not uncommon for them to dedicate more resources to the routing than for flow monitoring. Examples for limitations can be found in under-dimensioned flow caches or inaccurate time stamps [45, 58]. Router vendors typically employ methods of traffic sampling to cope with the high number of packets or flows in the network. If sampling is deployed on a flow-generating device, only a subset of all packets will be observed and accounted [59]. Hence, the flow data will only contain information about the subset of sampled packets. We will discuss sampling in more detail in Section 3.2.

Many commercial systems for working with flow data, especially with NetFlow [60] or IPFIX [55] exist. SWITCH maintains a list of tools and platforms that are capable of generating or processing NetFlow or IPFIX data [61]. A recent survey by Li et al. details many flow data analysis that researchers performed on flow data in the past [62]. We will use flow data in several parts of this thesis to analyze traffic volumes.

#### 2.1.4.3 Packet-Level Measurements

Packet-level measurements reveal most information about the traffic in a network. A traffic analysis system that performs packet-level measurements can observe the content of the traffic on all available layers[1]. Information from packet-level measurements can be aggregated to flow-level or link-level measurements.

Many tools for packet-level measurements exist for various purposes, such as security-related traffic analysis, flow generation, or service-level agreement monitoring on the application layer. The

1 Encryption makes it more difficult or impossible to extract information from the payload.

measurements are usually conducted by setting the network interface to *promiscuous mode* in order to receive all packets that pass the device, in particular including packets that are not directed to the interface.

Packet-level measurements can be conducted by devices that are connected to a span or mirroring port on a switch or router. Or the functionality can be built into the routers or switches directly, e.g. the NetFlow generation process in Cisco devices is a packet-level measurement that is built into the device. Traffic analysis applications that perform tasks like signature detection require full packets, i.e. including application layer payload. Other applications, such as flow generation, only need packet headers to conduct their work.

Like other types of passive measurements, packet-level analysis systems have to be placed at a location that allows them to observe traffic. Many DPI systems are located at the border(s) between the core network and the Internet, or between internal separated networks. Other good places can be near critical infrastructure or services in order to monitor their operation.

However, the links to the Internet are probably the most often used locations for the deployment of dedicated packet-level monitoring systems. Activities that can be observed at the border to the Internet cover attacks from the open Internet into the protected infrastructure as well as attacks or malicious traffic that originates from a compromised device within the own network. For example, infected machines within the protected network that participate in a botnet, usually connect to a ”Command & Control” (C&C) server in the Internet in order to receive commands. Since every connection from within the network to the outside Internet can be observed at the Internet border, a multitude of other malicious activities like scanning, spamming, or the infection of other machines can also be detected. Researchers proposed many approaches that use DPI systems for different tasks. We will discuss several security-related applications in Chapter 4.

Beside security monitoring, packet-level measurements are the basis for general network traffic analysis, or they are performed in order to gain application-specific insight. Several systems aim at providing traffic awareness to operators by analyzing and reporting information about traffic streams in the network. Ntop [63], for example, uses packet level measurements for network inventory, providing a recent view of active devices, employed protocols and communication patterns in the network. Researchers at the Politecnico di Torino created tstat [64, 65], a tool that examines network traffic for various purposes. It includes functions for general traffic analysis and traffic classification [66], and is able to analyze certain applications such as voice- and multimedia streaming services [67, 68].

The main focus of this thesis is on packet-level analysis systems.

## 2.2 Data Preparation and Normalization

Monitoring architectures often involve multiple data sources with access to log files and traffic measurements like the ones discussed before in this chapter. Networks often use products from different vendors to form a heterogeneous environment. Heterogeneous environments often use different protocols or log file formats depending on the vendors that supply equipment. Monitoring platforms need to cope with the diversity and either provide tools that are capable of analyzing the different formats or provide translation between data formats or data models. Many network monitoring platforms provide integration of different data sources into a single integrated system. A list of available platforms and tools can be found at [50].

## 2.3 Data Analysis

We discussed several data sources in this chapter. Many of them are located in different locations throughout the network. An analysis that makes use of the different data sources needs mechanisms for data fusion. Data must be either collected on a central system, or the analysis needs to be distributed throughout multiple locations. A central collection system can be overloaded with the overall amount of data. A distributed analysis system can be very complex and might require the exchange of large portions of information between its distributed modules. To the best of our knowledge, there does not exist a single system or single best practice that helps with designing a universal network monitoring or traffic analysis system. Solutions for certain applications have been proposed by researchers. In the field of security monitoring, for example, researchers presented ideas for distributed solutions for intrusion detection, attack mitigation or early warning systems such as the DIADEM Firewall [69], EMERALD [70] or CarmentiS [71].

Packet-level measurements are data sources that often provide large amounts of data. Packets are therefore often analyzed on the system that performs the measurements. Only the results of this analysis are then reported or transferred to another system. An example for such type of analysis and reporting is the generation of flow data: packets are the input to the process while flows are the result of the analysis.

However, approaches that distribute the packet-level measurements onto multiple machines exist as well. Schneider et al. [72], Kruegel et al. [73], or Colajanni et al. [74] present approaches for distribution of packets from a single observation point onto multiple machines. They use a cluster-setup to reduce the load on the individual analysis systems. This traffic distribution focuses on scenarios where no state must be exchanged between the analyzers, i.e. each of them can work independent from the others. Vallentin et al. discuss a distributed traffic analysis system that allows for collaboration between the analyzers [75]. The authors try to minimize the need for state exchange between the nodes, in order to avoid performance penalties.

Another approach that allows the distribution of packet-level analysis relies on sampling, e.g. as defined by PSAMP [59]. Such a setup has been proposed and implemented by Münz et al. in [76]. In their architecture, a packet-level measurement process generates flow data, which contains the payload of a carefully sampled set of packets. The flow data is then exported to a remote location that employs the analysis system TOPAS [77, 78]. TOPAS then extracts the packet payload content, creates new packet data and injects the crafted packets to a Snort detection process.

## 2.4 Result Reporting

Monitoring aims at the generation of information on a network. A human at the end of the monitoring architecture must often interpret the generated analysis results. The generated analysis results must often be interpreted by a human at the end of the monitoring architecture. For example, a monitoring process that targets the identification of attacks, infection of machines, or signs of botnet traffic, can generate a large number of alarms. Unfortunately, some of these alarms can be triggered by benign traffic, i.e. these are false positive alarms. Human operators eventually have to decide whether to take action or ignore the alarm. Presenting the analysis results to a human operator in a way that enables him to react on the alarm is important for many monitoring architectures.

The result presentation is also important for fields beyond security monitoring. If a network failure is detected, operators must be informed about the failure and must be given the possibility to identify the root cause of the problem. This allows them to decide on actions to resolve the problem.

Such analysis results can be presented in various ways. They can be presented in log files, reports that are sent via emails or accessible via web front ends. In the following, we will discuss several systems for the presentation of network flow data.

### 2.4.1 Visualization of Network Flow Data

**Notice of adoption from previous publication:** The text in this section is a modified version of the state of the art analysis of

- Lothar Braun, Mario Volke, Johann Schlamp, Alexander von Bodisco, and Georg Carle. "Flow-Inspector: A Framework for Visualizing Network Flow Data using Current Web Technologies," in First IMC Workshop on Internet Visualization (WIV 2012), Boston, MA, November 2012. [5]

The author of this thesis provided major contribution to the analysis of the state of the art.

The need for visualization of traffic data resulted in many systems with different analysis purposes. Some of them display information on traffic volumes and constituencies. NfSen [79] and

FlowScan [80] provide mechanisms for visualizing Netflow data that is generated by monitoring probes, routers, or switches. Their visualization methods concentrate on presenting traffic volumes including information about the used transport protocols.

Ntop [63] is a web-based system for analyzing and visualizing traffic information and also focuses on flow traffic analysis. It provides tools for collecting and generating flow information for its visualization process. In addition to standard volume-based visualizations, ntop provides tools for combining flow information with other data sources such as BGP data or IP-based geographical information.

Other approaches do not provide web interfaces for visualizations tasks. Instead, they created full-fledged client applications that perform the rendering. These, usually platform-depended applications, can make use of 3D capabilities of the client systems' graphic cards.

In [81], the authors present the client-based visualization tool FlowVis, which uses the SiLK tools for processing NetFlow data. They provide proof-of-concept visualizations like activity plots, flow edge bundles, and network bytes viewer. Lakkaraju et al. presented NVisionIP [82], a tool for displaying traffic patterns in class-B networks using scatter plots and volume-based visualizations. Yin et al. presented an animated link analysis tool for Netflow data [83], which leverages a parallel coordinate plot to highlight dependencies in the network.

We presented another approach for visualization of flow data called "Flow-Inspector". It is a web-based application that provides a Javascript-based web application, that differs from the other web-based approaches:

NfSen, FlowScan and ntop perform the visualization at server side, providing fixed images to the user. Flow-Inspector on the other hand renders its images in a client's browser. This allows for providing users with a higher level of interaction.

Concerning level of interaction, Flow-Inspector is comparable to other approaches that implement the visualization in an application that must be installed on the system. However, it has several advantages compared to those approaches. Due to its web-based nature, any computer with a modern browser can be used to interact with our system without any additional software installations required. Another major advantage is the extensibility for our work: An active community of JavaScript developers is working on visualization libraries for various purposes. Although such libraries might aim at implementing new visualization techniques for tasks beyond network flow analysis, corresponding approaches can often be adopted for displaying traffic data. Flow-Inspector users can benefit from such developments due to its extensible framework that allows to easily integrate these new visualization libraries.

# Part II

# Analysis of Related Work

# 3 Packet Capture Architectures and Performance

Traffic analysis systems can be decomposed into multiple parts. An important one is the process of packet capture. Packet capture covers all mechanisms that are in place to transport network packets from the network card to the analysis application. Standard network stacks found in today's general-purpose operating systems divide packet capture into several functions.

The performance of a capturing process depends on each of them to some extent. On the one hand, there is the hardware that needs to capture and copy all packets from the network to memory before the analysis can start. On the other hand, there is the driver, the operating system and traffic analysis application that need to carefully handle the available hardware in order to achieve the best possible packet capturing performance.

In the following, we will introduce popular capturing solutions and their components on Linux and FreeBSD in Section 3.1.1. Afterwards, we will summarize comparisons and evaluations that have been performed on the different solutions in Section 3.1.2.

## 3.1 Performance of Packet Capture Systems

**Notice of adoption from previous publication:** This study of related work presented in this section was performed in 2010, and describes the state-of-the-art in software at this time. The text is a revised version of Section 2 of

- Lothar Braun, Alexander Didebulidze, Nils Kammenhuber, and Georg Carle, "Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware," in Proceedings of the 10th Annual Conference on Internet Measurement (IMC '10), Melbourne, Australia, Nov. 2010." [2]

The author of this thesis conducted major parts of the research of the state of the art analysis. He provided significant input to the analysis of the software stack of the different operating systems and the proposed improvements. The text in this chapter received minor clarifications to the original version from the paper.

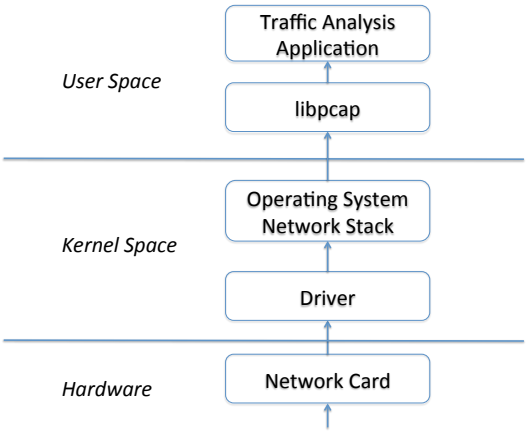



**Figure 3.1:** Subsystems involved in the capturing process

### 3.1.1 Solutions on Linux and FreeBSD

Advances made in hardware development in recent years such as high speed bus systems, multi-core systems or network cards with multiple independent reception (RX) queues offer performance that has only been offered by special purpose hardware some years ago. Meanwhile, operating systems and hardware drivers have come to take advantage of these new technologies, thus allowing higher capturing rates.

#### 3.1.1.1 Hardware

The importance of carefully selecting suitable capturing hardware is well known, as research showed that different hardware platforms can lead to different capturing performance. Schneider et al. [72] compared capturing hardware based on Intel Xeon and AMD Opteron CPUs with otherwise similar components. Assessing an AMD and an Intel platform of comparable computing power, they found the AMD platform to yield better capturing results. AMD's superior memory management and bus contention handling mechanism was identified to be the most reasonable explanation. Since then, Intel has introduced Quick Path Interconnect [84] in its recent processor families, which has improved the performance of the Intel platform; however, we are not able to compare new AMD and Intel platforms in this study due to lack of hardware. In any case, users of packet capturing solutions should carefully choose the CPU platform, and should conduct performance tests before buying a particular hardware platform.

Apart from the CPU, another important hardware aspect is the speed of the bus system and the used memory. Current PCI-E buses and current memory banks allow high-speed transfer of packets from the capturing network card into the memory and the analyzing CPUs. These hardware advances thus have shifted the bottlenecks, which were previously located at the hardware layer, into the software stacks.

#### 3.1.1.2 Software stack

Several software subsystems are involved in packet capture, as shown in Figure 3.1. Passing data between and within the involved subsystems can be a very important performance bottleneck that can impair capturing performance and thus lead to packet loss during capturing. We will discuss and analyze this topic in Section 5.3.3.

A packet's journey through the capturing system begins at the Network Interface Card (NIC). Modern cards copy the packets into the operating systems kernel memory using Direct Memory Access (DMA), which reduces the work the driver and thus the CPU has to perform in order to transfer the data into memory. The driver is responsible for allocating and assigning memory pages to the card that can be used for DMA transfer. After the card has copied the captured packets into memory, the driver has to be informed about the new packets through a hardware interrupt. Raising an interrupt for each incoming packet will result in packet loss, as the system gets busy handling the interrupts (also known as an *interrupt storm*). This well-known issue has lead to the development of techniques like interrupt moderation or device polling, which have been proposed several years ago [85, 86, 87]. However, even today hardware interrupts can be a problem because some drivers are not able to use the hardware features or do not use polling—actually, when we used the `igb` driver in FreeBSD 8.0, which was released in late 2009, we experienced bad performance due to interrupt storms. Thus, even though this issue is known to be a problem, bad capturing performance can still be explained by bad drivers; therefore, users should check the number of generated interrupts if high packet loss rates are observed[1].

Linux provides a technique for mitigating the interrupt load called NAPI (New API). The NAPI framework defers packet reception from the hardware interrupt handler to a later point in time. In the beginning, hardware interrupts are enabled. If a hardware interrupt occurs, then the normal operation of the system is suspended and the hardware interrupt handler is called. It is supposed to fulfill its task as fast as possible in order to allow the system to resume with normal operation. In order to achieve this goal, packet reception is deferred to a specific *poll()* function that must be implemented by the driver. The hardware interrupt only schedules a call to poll, which will be performed at a later point in time by the system. Furthermore, the hardware interrupt is expected to disable interrupts [88][2].

As soon as the *poll()* function is running, it passes the received packets into the network stack of the operating system. From there on, packets need to be passed to the monitoring application that wants to perform some kind of analysis. The standard Linux capturing path leads to a subsystem called PF_PACKET; the corresponding system in FreeBSD is called BPF (Berkeley Packet Filter). Improvements for both subsystems have been proposed.

1 FreeBSD will report interrupt storms via kernel messages. Linux exposes the number of interrupts via the proc file system in */proc/interrupts*.

2 Some drivers, like igb, use a hardware features called Interrupt Throttle Rate (ITR) to make the network card reduce the number of interrupts instead.

#### 3.1.1.3 Software improvements

One of the most prominent replacements for PF_PACKET on Linux is called PF_RING and was introduced in 2004 by Luca Deri [20]. Deri found that the standard Linux networking stack at that time introduced some bottlenecks, which lead to packet loss during packet capture. His capturing infrastructure was developed to remove these bottlenecks. He claimed to achieve higher capturing rates with PF_RING compared to PF_PACKET when small packets are to be captured. PF_RING ships with several modified network card drivers. These are changed to directly copy packets into PF_RING and therefore completely circumvent the standard Linux networking stack. This modification further boosts the performance for network cards that have the adapted drivers.

One important feature of PF_RING is the way it exchanges packets between user space and kernel: Monitoring applications usually access a library like `libpcap` [89] to retrieve captured packets from the kernel. Libpcap is an abstraction from the operating systems' capturing mechanisms and allows running a capturing application on several operating systems without porting it to the special capturing architecture. Back in 2004, the then current libpcap version 0.9.8 used a copy operation to pass packets from the kernel to the user space on Linux. An unofficial patch against that libpcap version from Phil Woods existed, which replaced the copy operation by a shared memory area that was used to exchange packets between kernel and application [90]. This modification will be called MMAP in the remainder of this work. PF_RING uses a similar structure to exchange packets by default. Libpcap version 1.0.0, which was released in late 2008, is the first version that ships built-in shared memory (SHM) exchange support; hence the patch from Phil Woods is not longer necessary. We will analyze the performance of these different solutions in Section 5.3.2.

All capturing mechanisms on Linux have something in common: They handle individual packets, meaning that each operation between user space and kernel is performed on a per-packet basis. FreeBSD packet handling differs in this point by exchanging buffers containing potentially several packets using a module called BPF, as shown in Figure 3.2.

Both BPF as well as its improvement Zero-Copy BPF (ZCBPF) use buffers that contain multiple packets for storing and exchanging packets between kernel and monitoring application. BPF and ZCBPF use two buffers: The first, called HOLD buffer, is used by the application to read packets, usually via libpcap. The other buffer, the STORE buffer, is used by the kernel to store new incoming packets. If the application (i.e., via libpcap) has emptied the HOLD buffer, the buffers are switched, and the STORE buffer is copied into user space. BPF uses a copy operation to switch the buffers whereas ZCBPF has both buffers memory-mapped between the application and the kernel. Zero-Copy BPF is expected to perform better than BPF as it removes the copy operation between kernel and application. However, as there are fewer copy operations in FreeBSD than in non-shared-memory packet exchange on Linux, the benefits between ZCBPF and normal BPF in FreeBSD are expected to be smaller than in Linux with shared memory support.

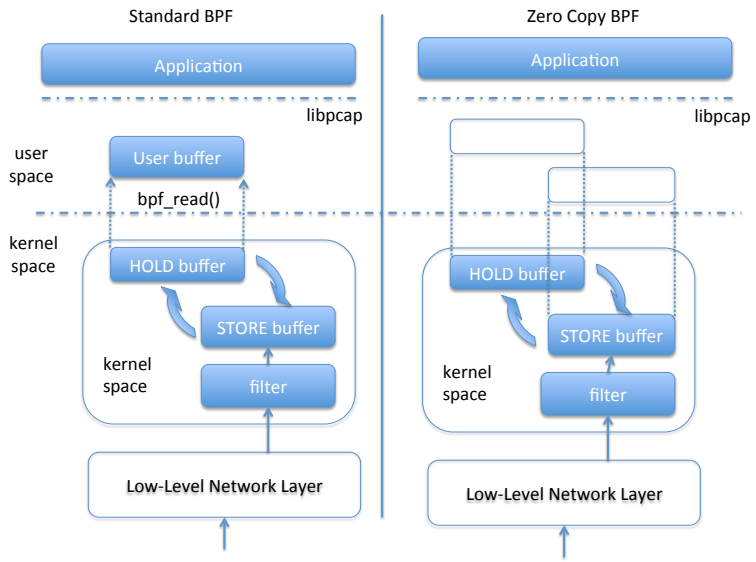



**Figure 3.2:** Berkeley Packet Filter and Zero Copy Berkeley Packet Filter

TNAPI is an improvement to the NAPI development by Luca Deri [91] for Linux. It introduces certain changes to standard Linux network card drivers and can be used in conjunction with PF_RING. Usually, Linux drivers assign new memory pages to network cards for DMA after the card copied new packets to old memory pages. The driver allocates new pages and assigns them to the card for DMA. Deri changed this behavior in two ways: NAPI drivers receive an interrupt for received packets and schedule a *poll()* call from the network stack to perform the packet processing. Deri's TNAPI driver creates a separate kernel thread that is only used to perform this processing. Hence, there is always a free kernel thread that can continue, and packet processing can almost immediately start.

The second major change is the memory handling within the driver: As soon as the thread is notified about a new packet arrival, the new packet is copied from the DMA memory area into the PF_RING ring buffer. Usually, network drivers would allocate new memory for DMA transfer for new packets. Deri's drivers allocate memory once when the driver is started. It then tells the card to reuse the old memory page, thus eliminating the necessity of allocating and assigning new memory.

Furthermore, his drivers may take advantage of multiple RX queues and multi-core systems. Modern network card offer a feature called *Message Signaled Interrupts* [92]. With this feature, they are able to deliver interrupts to specific cores. An RX queue of the NIC delivers its interrupts to a single core on the machine. TNAPI drivers create a kernel thread for each active RX queue of the card. Each kernel thread is bound to the CPU core that receives the interrupts for the queue.

A rather non-standard solution for wire-speed packet capture is *ncap*. Instead of using the operating system's standard packet processing software, it uses special drivers and a special capturing library. The library allows an application to read packets from the network card directly [93].

### 3.1.2 Previous Comparisons

Previous work compared the different approaches to each other. We will now summarize the previous findings and determine which experiments we need to repeat with our hardware platforms and new software versions. Results from related work can also give hints on further experiments we need to conduct in order to achieve a better understanding of the involved processes.

Capturing traffic in 1 GE networks is seen as something that today's off-the-shelf hardware is able to do, whereas it remains a very challenging task in 10 GE networks [72]. Apart from the ten-fold increased throughput, the difficulties also lie in the ten-fold increased packet rate, as the number of packets to be captured per time unit is a factor that is even more important than the overall bandwidth. As an example, capturing a 1 GE stream that consists entirely of large packets, e.g., 1500 bytes, is easy; whereas capturing a 1 GE stream consisting entirely of small packets, e.g., 64 bytes, is a very difficult task [20, 72]. This is due to the fact that each packet, regardless of its size, introduces a significant handling overhead.

Driver issues that arise with a high number of packets have been studied very well [85, 86, 87]. Problems concerning interrupt storms are well understood and most network cards and drivers support mechanisms to avoid them. Such mechanisms include polling or interrupt moderation.

In 2007, Schneider et al. compared FreeBSD and Linux on Intel and AMD platforms [72]. They determined that device polling on Linux reduces the CPU cycles within the kernel and therefore helps to improve capturing performance. On FreeBSD however, device polling actually reduced the performance of the capturing and furthermore reduced the stability of the system. Hence, they recommend using the interrupt moderation facilities of the cards instead of polling on FreeBSD. In their comparison, FreeBSD using BPF and no device polling had a better capturing performance than Linux with PF_PACKET and device polling. This trend is enforced if multiple capturing processes are simultaneously deployed; in this case, the performance of Linux drops dramatically due to the additional load. Schneider et al. find that capturing 1 GE in their setup is possible with the standard facilities because they capture traffic that contains packets originated from a realistic size distribution. However, they do not capture the maximum possible packet rate, which is about 1.488 million packets per second with 64 bytes packets [91]. Another important aspect about the workload of Schneider et al. is that during each measurement, they send only one million packets (repeating each measurement for 7 times). This is a very low number of packets, considering that using 64 byte packets, it is possible to send 1.488 million packets within one second. Some of the effects we could observe are only visible if more packets are captured. Based on their measurement results, they recommend a huge buffer size in the kernel buffers, e.g., for the HOLD and STORE buffers in FreeBSD, to achieve good capturing performance. We can see a clear correlation between their recommendation and the number of packets they send per measurement and will come back to this in Section 5.3.2.

Deri validated PF_RING against the standard Linux capturing PF_PACKET in 2004 [20]. He finds PF_RING to really improve the capturing of small (64 bytes) and medium (512 bytes) packets compared to the capturing with PF_PACKET and libpcap-mmap. His findings were reproduced by Cascallana and Lizarrondo in 2006 [94] who also found significant performance improvements with PF_RING. In contrast to Schneider et al., neither of the PF_RING comparisons considers more than one capturing process. Using TNAPI and the PF_RING extensions, Deri claims to be able to capture 1 GE packet streams with small packet sizes at wire-speed (1.488 million packets) [91].

In this thesis, we develop and implement a testing environment to compare the performance of the different systems. We provide evaluations of the different technologies using different scenarios with multiple application-level workloads. Our setups and findings are detailed in Chapter 5.

## 3.2 Sampling Algorithms

Resource exhaustion on traffic analysis systems have been an issue for a long time. As early as 1986, Minnich described the challenges of building a packet capture system, and highlighted the performance problems of general purpose operating systems with respect to packet capture [95].

Sampling algorithms have therefore been the target of research for a very long time. Amer and Cassel provided an overview on statistically sampling measurements for computer networks in 1989 [96]. The scientific community provided multiple sampling algorithms throughout the years. The measurement community even decided to standardize traffic sampling for IP packet networks [97].

Many of these algorithms aim at extracting statistical properties of the traffic, e.g. estimates of the number of packets, bytes or flows. Zseby provided a very good overview on sampling research in her PhD thesis [98], and proposed sampling mechanisms for flow volume estimation and QoS measurement applications.

This work also considers sampling for traffic analysis tasks in overload scenarios. It focuses on DPI applications that aim at identifying very specific pieces of information from payload: security related information in Chapter 8, information about video download traffic in Section 9, and SSL/TLS certificates from TCP connections in Chapter 10. In contrast to applications that aim at information about the overall population of the traffic, the aforementioned traffic analysis tasks aim at very specific subsets of the traffic. Algorithms must carefully select the packets that contain the information that the analysis cares for, e.g. the sampling algorithms must pick the packets that contain SSL/TLS certificates. As every connection can potentially carry TLS/SSL traffic, we need to inspect each connection in order to determine whether it is a TLS/SSL connection or not. A sampling algorithm that fits our needs therefore needs to select traffic from every connection. It is well known that random sampling algorithms have problems with catching all flows in the

network [99]. In the following, we examine analysis methods and sampling methods that focus on packet or payload from the beginning of connections.

### 3.2.1 Sampling of the First N Bytes of Traffic Flows

**Notice of adoption from previous publication:** The text in this section is a slightly modified version of Section 2 of

- Lothar Braun, Gerhard Münz, and Georg Carle, "Packet Sampling for Worm and Botnet Detection in TCP Connections," in 12th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2010), Osaka, Japan, Apr. 2010. [1]

The author of this thesis contributed significantly to the analysis of the state of the art. The other parts of the paper, including an explanation of the contributions of the author, can be found in Chapter 6.

The importance of the first packets of a flow has been shown in the context of attack detection and traffic classification. Wang et al. present PAYL, a system that searches for anomalies in the payload of network packets in order to detect attack traffic [100]. An evaluation based on real traffic traces shows that the detection rate does not decrease significantly if only the first 1000 bytes of payload of each flow are analyzed instead of complete flows. On the other hand, processing time drops dramatically due to the reduced amount of analyzed traffic. Regarding traffic classification, Sen et al. are able to classify a flow as belonging to a P2P application after looking at the first ten packets [101]. Won et al. show that most application signatures appear within the first five packets of a flow [102]. The authors ascertain that analyzing all packets of the flow may even lead to misclassifications that do not occur if only the first packets are examined.

Several other works analyzed the importance of the first few packets for behavioral traffic classification. Berlaille et al. show that packet lengths and direction of the first four packets are a good metric for application classification [103]. Their approach has been improved by using Markov models on the same metrics by Münz et al. [104, 105].

Kornexel et al. introduced the *Time Machine* [106], which allows storing network traffic from high-speed networks for a longer time period. In order to record large numbers of flows with limited storage capacity, only a few kilobytes from the beginning of each (unidirectional) flow are saved. This significantly reduces the amount of disk space due to the heavy-tailed flow length distribution. The authors state that most of the relevant information necessary for security forensic is preserved in the retained data, yet no evidence is provided that this assumption is valid. Maier et al. propose to combine the *Time Machine* with an intrusion detection system in order to correlate ongoing traffic with past observations [107]. The authors can show that they can improve the detection rates of an Intrusion Detection System by using historical data that contains the start of biflows.

This thesis will state the applicability of sampling of the first payload bytes of each biflow for additional traffic analysis tasks in Part IV of the document. Furthermore, we present an algorithm that performs static sampling of the first N bytes of each TCP connection using Bloom Filters in Chapter 6.

The utilization of Bloom filters [26] and similar data structures has been proposed for various traffic measurement purposes. Chang et al. use Bloom filters to store recent packet classification results [108]. Kong et al. use Time-out Bloom filters (i.e., Bloom filters saving time stamps) to sample the first packet of every observed flow [109]. The same authors combine the Time-out Bloom filter with a Count-Min Sketch (CMS), which is a Bloom filter storing counters, to detect port scans [110]. The usage of CMS has also been proposed for counting the number of distinct flows [111, 112], for counting the number of packets or bytes per flow [113, 114, 115], and for distributed traffic monitoring [116].

Whitehead et al. memorize every observed TCP SYN packet in Bloom filters to detect established TCP connections [117]. Only packets belonging to an established TCP connection are sampled, yet without limiting the number of sampled packets. Close to our approach is the work of Canini et al. who use a chain of Bloom filters to sample the first $J$ packets of every bidirectional flow [118]. The sampling limit is expressed as the maximum number of packets, which works fine for classifying most kinds of legitimate traffic but makes it easy for an attacker to circumvent packet selection, for example by sending empty TCP ACK packets.

Our packet sampling algorithm aims at sampling the first $N$ payload bytes of each TCP connection and thus approximates the behavior of the *Time Machine* [106]. In contrast to the *Time Machine* and Canini's approach, our algorithm tracks the establishment and termination of every TCP connection, which enables us to remove the saved information of terminated connections. Hence, we do not need to reset the Bloom filters periodically. Moreover, we demonstrate the usability of our sampling strategy for signature-based worm and botnet detection by examining real worm and botnet traffic in Chapter 8.

### 3.2.2 Adaptive Sampling of the Flow Start Bytes

**Notice of adoption from previous publication:** The text in this section contains parts of Section II of the previous publication:

- Lothar Braun, Cornelius Diekmann, Nils Kammenhuber, Georg Carle, "Adaptive Load-Aware Sampling for Network Monitoring on Multicore Commodity Hardware," in Proceedings of the IEEE/IFIP Networking 2013, New York, NY, May 2013. [6]

The author of this thesis provided major contributions to the analysis of the state of the art. This presentation of the related work in this section includes works that were not discussed in the paper.

Although previous research and this thesis finds that sampling of the first *N* bytes of traffic is a very promising sampling approach, selecting an appropriate value for the per-flow sampling limit *N* is difficult: if *N* is set too low, many interesting packets will be filtered out by the sampling process; whereas if *N* is too high, system resources will be insufficient to handle all the sampled packets, and packet loss is inevitable. Either way, interesting packets can go undetected which could have been found with a better value for *N*. For attack detection, a major drawback of the approaches is the fact that a malicious attacker can try to send *N* bytes of legitimate traffic before sending attack packets, thus evading detection (see Chapter 8).

The problem of determining an optimal flow sampling limit *N* is challenging due to the fact that network traffic tends to undergo dramatic fluctuations, in volume as well as in traffic mixture, on comparably short timescales. These traffic properties result in different traffic features, which in turn change the maximum number of bytes per flow that an intrusion detection or protocol parsing system can handle. We therefore aim in this thesis to further extend the promising sampling idea and propose a sampling algorithm that selects the first *N* bytes, but with *N* being continuously and dynamically adjusted to reflect the current network and analysis workloads in Chapter 7. To the best of our knowledge, nobody so far proposed a sampling algorithm that selects traffic from the beginning of bi-flows and adapts the number of sampled bytes according to the needs of the target traffic analysis application.

However, adaptive algorithms have been studied with other applications in mind. Jurga and Hulboj provide an overview of adaptive sampling algorithms in a technical report [119]. They survey a number of adaptive sampling algorithms and very briefly discuss their adaptation mechanisms. Adaptive algorithms have been proposed for different goals in mind. Most commonly, they provide an adaptation in order to cope with computational resource limitations by including the resources into the adaptation process. Others do not explicitly model resource limitation, but try to minimize the amount of sampled packet according to the needs of the target traffic analysis application.

Drobisz and Christensen present a adaptive sampling algorithm for the calculation of network statistics [120]. They show that an adaptive sampling algorithm can provide more accurate estimates for packet count means, variances, and the Hurst parameter [121]. The authors extend a static sampling proposed by Claffy et al. [122], by modeling CPU utilization and adapting the sampling rate depending on utilization thresholds.

Choi et al. propose an adaptive sampling algorithm for enhanced measurement accuracy [123, 124]. The algorithm divides the time into discrete intervals, and estimates the number of packets and the squared coefficient of variation (SCV) of packet sizes. A new packet sampling probability is calculated for the next time interval based on the packet count estimate of the previous intervals. The authors employ an Auto-Regressive model to get good new estimates from the previous measurement intervals.

Zhang et al. [125] presented a sampling algorithm that is botnet-aware and tries to sample traffic that likely belongs to a botnet, such as Command and Control traffic. The authors embed parts of the botnet detection logic into the sampling algorithm in order to only sample packets from IPs that show suspicious behavior. Suspicious IPs are determined by synchronized operations throughout single time bins as well as over multiple time epochs using cross-epoch correlation. Their approach is very application-specific and does not adapt depending on packet consumption rates.

Patcha et al. present an adaptive sampling algorithm for Denial-of-Service (DoS) attack detection [126]. They argue that a key indicator for detecting DoS attacks is to detect the absence of self-similarity in network traffic. Their algorithm aims at sampling packets in a way that conserves the self-similar properties in the sample. Therefore, time intervals are defined and sampling rates are adopted for each interval. A history of the past sampling intervals are used to predict a good value for the next interval using weighted least squares predictor. Resource constraints are not taken into account for the determination of the sampling probability.

Hernandez et al. present two adaptive sampling algorithms for network management for time-based sampling for SNMP-based measurements [127]. It aims at providing robust network awareness in the events of high network usage by adapting the sampling interval: When high level of activity on the network is detected, the algorithm shortens the sampling interval in order to provide more accurate measurements. Low activity levels stretch the sampling interval, thus reducing the sampling overhead. The first proposed algorithm uses linear prediction, the second uses fuzzy logic to adjust the sampling rate.

Estan et al. [99] propose Adaptive NetFlow, an extension to NetFlow that includes adaptions of the sampling rate depending on available computational resources. They consider memory and CPU resources as crucial resources that need to be protected throughout the measurement process by adapting the sampling rate. A fixed time interval has to be defined by the user along with a target number of flows that should be sampled for each measurement interval. Adaptive NetFlow then dynamically adapts the sampling rate to retrieve the desired number of flows and keep resource consumption constant.

Barlet-Ros et al. [128] propose a system that adapts the sampling rate based on the needs of the monitoring application. Their system observes traffic features and tries to determine their impact on monitoring application by monitoring its CPU usage. Incoming packets are grouped into batches, and several traffic features are calculated for each batch. A prediction model is used to estimate the number of CPU cycles required to process the packet batch based on the observation of past batches. If the predicted number of CPU cycles exceeds the number of available CPU cycles, sampling is applied to the packet batch. The sampling rate is calculated based on the factor that the predicted number of CPU cycles exceeds the available number of CPU cycles.

Our approach differs from the previous work in one or more of the following points: First, we do not examine the number of CPU cycles or indicators that other system resources are exhausted. We

focus on the observation of a single metric that reflects application load and incoming traffic in an indirect way: the fill level of the buffer between the network stack and the analysis application. We therefore do not care which resources is exhausted, e.g. CPU or memory, but only observe if an application is capable to consume all the traffic. Second, we adopt the sampling rate continuously on each packet arrival. Some of the other approaches define time windows and adopt the sampling rate at the end of the time window. This allows for faster adoptions of the sampling rate in case of short-lived events. Last but not least, most other approaches directly adopt the sampling *rate*, while we change the sampling *limit per bi-flow*. Sampling rate and sampling limit are, in general, not connected linearly, but depend on the flow length distribution.

# 4 Traffic Analysis

Part IV of the thesis focuses on applications of traffic analysis systems, as well as on reporting the results of the analysis processes to human operators. During this process, we use traffic measurements for the tasks presented in the introduction of this thesis: analysis of video traffic for caching benefits, security monitoring, and analysis of the currently deployed TLS/SSL PKI infrastructure.

Security related traffic analysis processes that are used to detect botnets and malicious botnet-related traffic are discussed in Section 4.1. Section 4.2 discusses related work on YouTube's video infrastructure studies and caching potentials that we build on in our work. The studies on the SSL landscape that pre-dates our analysis are discussed in Section 4.3.

## 4.1 Botnet Detection in Early Packets

As previously described in Section 3.2.1, a number of work focused on the detection of malicious traffic in the first few packets [100, 101, 102, 106, 107]. However, most approaches for botnet detection assume that all the available traffic can be analyzed and do not make assumptions as to where within a TCP connection or biflow malicious content can be found.

Wurzinger et al. describe a way to automatically infer Bro signatures for botnet detection from dynamic malware analysis results [129]. They propose to automatically extract signatures for the Bro-IDS from real botnet traffic collections. As a first step, malware is collected and executed for a period of several days in a controlled environment. The malware is allowed to send its Command & Control (C&C) traffic into the Internet. All outgoing and incoming traffic is recorded for the automatic signature extraction process. Change point detection is used to identify "relevant" traffic patterns. These patterns will be extracted and collected in a pool of relevant patterns. The basic idea is that important patterns will be seen often and can therefore be automatically identified, e.g. a bot master issues a command to its botnet, which results in scanning behavior. Hierarchical clustering is used to create behavior clusters that contain traffic patterns of similar behavior, e.g. command and scanning, command and spamming, command and denial of service. Using a longest common sequence algorithm, signatures can be created that match for the bot masters commands, or the bot responses. The command and response signatures can be inferred for arbitrary C&C protocols, e.g. IRC, HTTP, or P2P, even if the used protocol is unknown, e.g. if a custom C&C protocol

was developed by the botnet author. The generated signatures can be loaded into the Bro-IDS and can be then be used for online botnet detection. While they do not explicitly discuss where these signatures are found within connections, they report on rules that match to the beginning of incoming packets.

Similar work has been done by Rieck et al. [130], who also automatically generate signatures for botnet traffic detection from malware sample traffic. They performed their work independently from us and published their results one month before our publication on NOMS 2010. Rieck et al. generate their signatures such that they match within the first few bytes of biflows, and they build a detection system that specifically performs TCP reassembling of the first few TCP bytes. The evaluation of their system shows that they can reliably detect the traffic of the bots from their samples within the first 256 bytes of each biflow. This finding is linked to the process of rule generation: The rule generation process only takes the first bytes of each flows into account, thus producing rules that are specific for the beginning of the flows. It is unclear from their analysis whether this good detection performance in the beginning of the flows could be achieved with the Snort standard rules or specialized rules sets such as the one provided by EmeringThreats [131].

Münz et al. provide some insight into this question in their 2007 paper [76]. The authors examine the Snort rule sets provided by the Sourcefire as part of the Snort distribution. They find that many rules that specify payload patterns provide some means as to which bytes within a packet or stream the rule could match. As little as 145 bytes per packets are necessary to be enough payload 90% of all rules. Due to their rule analysis, the authors decide to use a sampling mechanism that exports the first 145 bytes of every packet.

This sampling mechanism is tested against the packet trace of the DARPA Intrusion Detection Evaluation 2000, DDos scenario 2.0.2, inside traffic [132]. The authors find that their sampling mechanism reduces the amount of traffic that must be analyzed by 66%. Snort rules that where applied on the sampled traffic hit 87.2% of all the alarms that would have been found if the rules had been applied to the total traffic.

Sampling every packet with a cutoff at some payload byte has the potential to sample more of the traffic than a sampling algorithm that selects the beginning of each flow. Kornexl et al. report on the amount of sampled traffic for various flow cutoff values [106]. Their evaluation results show that sampling the beginning of flows results in very large reductions of the overall traffic due to the heavy-tailed nature of the flow-length distribution. If general Snort rules apply to the beginning of flows, then there is an expected larger reduction in traffic that must be analyzed. This would lead to an even further load relieve on the traffic analysis system.

Another critique on their evaluation is the use of the DARPA data set. Sommer and Paxson, for example, argue against the use of this data set for evaluation of anomaly detection systems [133]. First, they criticize that the data set was created artificially, and contains a set of unfortunate artifacts. Second, the data set has been created in the late 90s, before the threat of botnets became

eminent. The evaluation therefore does not contain data on a security risk that is interesting for many operators.

In this thesis, we discuss the matching of attack detection on rules for Snort on botnet traffic and live traffic from real work in Chapter 8. We therefore use a data set that contains the real botnet traffic that has been collected using dynamic malware analysis tools, and we evaluate live traffic data from a 10GE university network tap.

## 4.2 Analysis of the YouTube Video Infrastructure

**Notice of adoption from previous publication:** The text in this section was adopted from Section II of the paper

- Lothar Braun, Alexander Klein, Georg Carle, Helmut Reiser, and Jochen Eisl. "Analyzing Caching Benefits for YouTube Traffic in Edge Networks - A Measurement-Based Evaluation," in Proceedings of the 13th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2012), Maui, Hawaii, April 2012. [4]

The author of this thesis conducted major parts of the research of the state of the art. The other parts of the paper, including an explanation of the contributions of the author of this thesis, can be found in Chapter 9.

Related work can be grouped into several categories: Some papers discuss on the shares of YouTube traffic in the overall traffic mix. Others focus on YouTube traffic characteristics, YouTube's infrastructure, or caching of video content.

Popularity of YouTube videos has been studied from several different points of view, (e.g. [134, 135, 136, 137, 138]). One branch of papers use active crawling of YouTube sites to determine popularity or try to find reasons for popularity of various videos [134]. Figueiredo et al. [139] focus on popularity development over time. Their findings conclude that a lot of videos show a *viral* popularity growth, indicating potentials for caching. Others find power-law patterns with truncated tails in global video popularity distributions and predict good caching potentials [140, 137].

Video popularity has also been studied from network local perspectives [136], and local and global video popularity have also been compared [135]. These studies show that the global popularity of video content does not have to match the local popularity. For example, Gill et al. [135] find that the Top 100 videos from global YouTube video rankings are viewed in their network but do not have any significant contribution to the overall amount of YouTube traffic observed. Caching strategies must therefore consider local popularity and view counts.

Other work tries to provide a better understanding of the YouTube web application or YouTubes' infrastructure. Such work includes attempts to inspect YouTubes' policies for picking local data

centers for video downloads [141], or describe load-balancing or traffic asymmetry from the view point of a Tier-1 provider [142]. Finamore et al. [67] assess YouTube traffic by evaluating several YouTube traces from different networks. They study traffic and video patterns for YouTube videos from mobile and PC-based devices, and show differences between traffic of these devices.

Ager et al. considered caching potentials for different protocols in [143] where they outline good potentials for caching HTTP in general. Zink et al. evaluate caching of YouTube videos [136]. In their study, the authors collect three one-week traces from a university network and use these as input for cache simulation. The authors consider client, P2P and proxy caches and estimate cache video hit rates. They conclude that high video hit rates can be achieved even with small caches.

We extend their work by accounting more important factors such as video encoding formats, aborted video downloads and their impact on caches. Furthermore, we do not only consider video hit rates, but more sophisticated metrics such as content hit rates. Using these metrics, we can show that other caching strategies, such as chunk-wise caching strategies, provide better cache performance than the previously proposed and evaluated caching. Our work reveals that video hit rates might not be a good metric in measuring video cache performances, and furthermore shows that simple non-video aware caching strategies can have negative impact on cache performance.

## 4.3 SSL Analysis

**Notice of adoption from previous publication:** The text in this section is based on the analysis of related work published in

- Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. “The SSL Landscape - A Thorough Analysis of the X.509 PKI Using Active and Passive Measurements” in Proceedings of the 11th Annual Internet Measurement Conference (IMC ’11), Berlin, Germany, November 2011. [3]

It summarizes the state of the art up to the publication in 2011. The author of this thesis contributed to the analysis of the state of the art.

Two major previous works have been published prior to our own paper at IMC 2011. Both studies on the TLS/SSL landscape were presented while our own measurement activities were still ongoing. They report on data sets that were obtained throughout our own measurement period.

The Electronic Frontier Foundation (EFF) presented a PKI architecture analysis at DEFCON 2010 [144] and 27C3 [145]. Both publications where given at hacker meetings and were published as slide sets. The EFF data was made available to allow other researchers to build on their data sets. They contain TLS/SSL certificates from active measurements that involved the full IPv4 address space. This measurement connected to the HTTPs port on the IP addresses and tried to conduct

a handshake. Two measurement runs were performed, the first between April and July 2010, the second in August 2010. The presentations of the analysis results focused on the CA structure, namely on the number and role of the Certification Authorities. Furthermore, the presenters reported on obvious errors or unexpected findings in the collected certificates.

Our measurement technique differs significantly from the one employed by the EFF (compare Chapter 10). Hence, we expect differences in the comparison between our data sets and the EFF data sets. Fortunately, the EFF published their data sets, which enabled us to include their data into our own analysis.

The second presentation was given at BlackHat 2010 [146] and Infosec 2011 [147] by Ivan Ristic. His presentation was based on results from active measurements that were conducted in July 2010. In the presentation at Infosec 2011, he compared his data with the one published by the EFF. Ristic's measurement is comparable to our own measurement technique: He employs a large number of domain names, including the content of the Alex Top 1 million list [148]. His active measurements queried the domains on his list for open HTTPs ports and performed a TLS/SSL handshake.

Compared to Ristic and the EFF, our own work differs in a number of points: First, we observe the state of the TLS/SSL deployment over an extended period of time, instead of reporting on a single deployment snapshot. Furthermore, our scans are conducted from multiple locations all over the world. These differences allow us to estimate how users experience the TLS/SSL-secured infrastructure in other parts of the world. We were also able to include the data from the EFF scans into our own data sets, thus allowing us to analyze their data with our own algorithms.

Most important, we also obtain certificates from passive traffic measurements in a large university network. We monitor all TLS/SSL connections between the network and the Internet. This allows us to enhance our evaluation of the infrastructure with data about the actual use of the infrastructure. Including this data into the evaluation also allows us to observe uses of TLS/SSL outside the closed HTTPs world.

# Part III

# Evaluation and Improvement of Traffic Analysis Systems

# 5 Performance of Packet Capture Systems

**Notice of adoption from previous publication:** The text in this chapter is a revised version of

- Lothar Braun, Alexander Didebulidze, Nils Kammenhuber, and Georg Carle, "Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware," in Proceedings of the 10th Annual Conference on Internet Measurement (IMC '10), Melbourne, Australia, Nov. 2010." [2]

The author of this thesis provided major contributions to the design of the tests that were used to evaluate the performance of the packet capture systems. He provided major contributions to the development of the *packzip* test tool that was used as part of the evaluation tests. The test setup was implemented as part of the master thesis of Alexander Didebulidze [149], which was supervised by the author of this thesis. The author of this thesis contributed significantly to the interpretation and analysis of the measurement results, to the proposed improvements and the recommendation to users and developers.

## 5.1 Introduction

Packet capture is an essential part of most network monitoring and analyzing systems. A few years ago, using specialized hardware—e.g., network monitoring cards manufactured by Endace [150]—was mandatory for capturing Gigabit or Multi-gigabit network traffic, if little or no packet loss was a requirement. With recent development progresses in bus systems, multi-core CPUs and commodity network cards, nowadays off-the-shelf hardware can be used to capture network traffic at near wire-speed with little or no packet loss in 1 GE networks, too [151, 93]. People are even building monitoring devices based on commodity hardware that can be used to capture traffic in 10 GE networks [72, 91]

However, this is not an easy task, since it requires careful configuration and optimization of the hardware and software components involved—even the best hardware will suffer packet loss if its driving software stack is not able to handle the huge amount of network packets. Several subsystems including the network card driver, the capturing stack of the operating system and the monitoring application are involved in packet processing. If only one of these subsystems faces performance problems, packet loss will occur, and the whole process of packet capturing will yield bad results.

Previous work analyzed [151, 72, 94] and improved [91, 85, 20] packet capturing solutions. Comparing these work is quite difficult because the evaluations have been performed on different hardware platforms and with different software versions. In addition, operating systems like Linux and FreeBSD are subject to constant changes and improvements. Comparisons that have been performed years ago therefore might today not be valid any longer. In fact, when we started our capturing experiments, we were not able to reproduce the results presented in several papers. When we dug deeper into the operating systems' capturing processes, we found that improved drivers and other improvements in the operating system can explain some of our results. Other differences can be explained by the type of traffic we analyzed and by the way our capturing software works on the application layer.

While it is not a problem to find comparisons that state the superiority of a specific capturing solution, we had difficulties to find statements on why one solution is superior to another solution. Information about this topic is scattered throughout different papers and web pages. Worse yet, some information proved to be highly inconsistent, especially when from Web sources outside academia. We therefore encountered many difficulties when debugging the performance problems we ran into.

This thesis tries to fill the gap that we needed to step over when we set up our packet capturing environment with Linux and FreeBSD. We evaluate and compare different capturing solutions for both operating systems, and try to summarize the pitfalls that can lead to bad capturing performance. Our work aims at future developers and users of capture systems and serves as a resource helping them not to repeat the pitfalls we and other researchers have encountered.

A special focus of this work is on explaining our findings. We try to identify the major factors that influence the performance of a capturing solution, providing users of packet capture systems with guidelines on where to search for performance problems in their systems. Our work also targets developers of capturing and monitoring solutions: We identify potential bottlenecks that can lead to performance bottlenecks and thus packet loss. Finally, we propose a modification that can be applied to popular capturing solutions. It improves capturing performance in a number of situations.

The remainder of this chapter is organized as follows: Section 5.2 presents the test setup that we used for our evaluation in Section 5.3. Our capturing analysis covers scheduling issues in Section 5.3.1 and focuses on the application and operating system layer with low application load in Section 5.3.2. Subsequently, we analyze application scenarios that pose higher load on the system in Section 5.3.3, where we furthermore present our modifications to the capturing processes and evaluate their influence on capturing performance. In Section 5.3.4, we move downwards within the capturing stack and discuss driver issues. Our experiments result in recommendations for developers and users of capturing solutions, which are presented in Section 5.4. Finally, Section 5.5 concludes the paper with a summary of our findings.

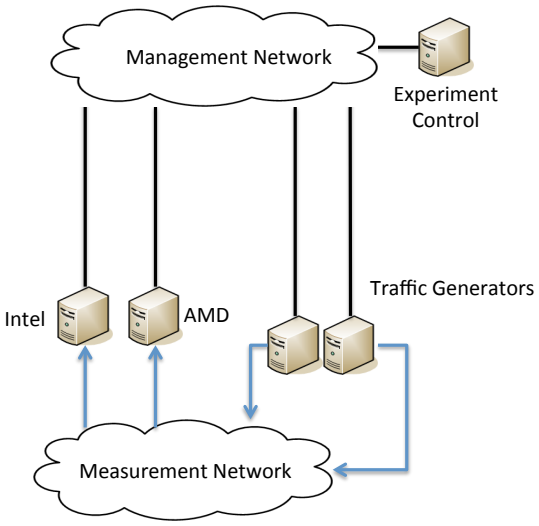



**Figure 5.1:** Evaluation Setup

## 5.2 Test setup

In this section, we describe the hardware and software setup that we used for our evaluation. Our test setup (see Figure 5.1) consists of five PCs, two for traffic generation, the other two for capturing, and a fifth machine for experiment control.

The traffic generators where equipped with several network interface cards (NICs) and uses the Linux Kernel Packet Generator [152] to generate a uniform 1 GE packet stream. It was necessary to generate the traffic using several NICs simultaneously because the deployed cards were not able to send out packets at maximum speed when generating small packets. Even with this setup one traffic generator was only able to generate 1.27 million packets per seconds (pps) with the smallest packet size. However, this packet rate was sufficient to show the bottlenecks of all analyzed systems. We had to deploy the second generator for our experiments in Section 5.3.4, where we actually needed wire-speed packet generation (1.488 million pps).

In contrast to [72], we did not produce a packet stream with a packet size distribution that is common in real networks. Our goal is explicitly to study the behavior of the capturing software stacks at high packet rates. As we did not have 10 GE hardware for our tests available, we had to create 64 bytes packets in order to achieve a high packet rate for our 1 GE setup.

Each of our test runs is configured to produce a packet stream with a fixed number of packets per seconds and runs over a time period of 100 seconds and is repeated for five times. As our traffic generator is software-based and has to handle several NICs, the number of packets per second

is not completely stable and can vary to some extend. Hence, we measure the variations in our measurements and plot them where appropriate.

Capturing is performed with two different machines with different hardware in order to check whether we can reproduce any special events we may observe with different processor, bus systems and network card, too. Two Intel Xeon CPUs with 2.8 GHZ each operate the first capturing PC. It has several network cards including an Intel Pro/1000 (82540EM), an Intel Pro /1000 (82546EB) and a Broadcom BCM5701 all connected via a PCI-X bus. In our experiments, we only use one NIC at a time.

The second capturing PC has an AMD Athlon 64 X2 5200+ CPU and is also equipped with several network cards, including an Intel Pro/1000 (82541PI) and a nVidia Corporation CK804 onboard controller. Both cards are connected via a slow PCI bus to the system. Furthermore, there are two PCI-E based network cards connected to the system. One is an Intel Pro/1000 PT (82572EI), the other is a Dual Port Intel Pro/1000 ET(82576). It should be noted that the AMD platform is significantly faster than the Xeon platform.

Using different network cards and different architectures, we can check if an observed phenomenon emerges due to a general software problem or if it is caused by a specific hardware or driver. Both machines are build from a few years old and therefore cheap hardware, thus our machines are not high end systems. We decided not to use more modern hardware for our testing because of the following reasons:

- We did not have 10 GE hardware available at the time of these experiments..
- We want to identify problems in the software stack that appear when the hardware is fully utilized. This is quite difficult to achieve with modern hardware on a 1 GE stream.
- Software problems that exist on old hardware, which monitors 1 GE packet streams, still exist on newer hardware that monitors a 10 GE packet stream.

Both machines are installed with Ubuntu Jaunty Jackalope (9.04) with a vanilla Linux kernel version 2.6.32. Additionally, they have an installation of FreeBSD 8.0-RELEASE for comparison with Linux.

We perform tests with varying load at the capturing machines' application layer in order to simulate the CPU load of different capturing applications during monitoring. Two tools are used for our experiments:

First, we use tcpdump 4.0.0 [153] for capturing packets and writing them to `/dev/null`. This scenario emulates a simple one-threaded capturing process with very simple computations, which thus poses almost no load on the application layer. Similar load can be expected on the capturing thread of multi-threaded applications that have a separate thread that performs capturing only. Examples for such multi-threaded applications are the Time Machine [106, 107] or the network monitor VERMONT [57].

The second application was developed by us and is called *packzip*. It poses variable load onto the thread that performs the capturing. Every captured packet is copied once within the application and is then passed to `libz` [154] for compression. The user can configure the compression mode from 0 (no compression) to 9 (highest compression level). Increasing the compression level increases CPU usage and thus can be used to emulate an increased CPU load for the application processing the packets. Such packet handling has been performed before in [151] and [72]. We used this tool in order to make our results comparable to the results presented in this related work.

## 5.3 Evaluation

This section presents our analysis results of various packet capture setups involving Linux and FreeBSD, including a performance analysis of our own proposed improvements to the Linux capturing stack. As multi-core and multi-processor architectures are common trends, we focus in particular on this kind of architecture. On these hardware platforms, scheduling issues arise when multiple processes involved in packed capturing need to be distributed over several processors or cores. We discuss this topic in Section 5.3.1. Afterwards, we focus on capturing with low application load in Section 5.3.2 and see if we can reproduce the effects that have been observed in the related work. In Section 5.3.3, we proceed to capturing with higher application load. We identify bottlenecks and provide solutions that lead to improvements to the capturing process. Finally, in Section 5.3.4 we present issues and improvements at the driver level and discuss how driver improvements can influence and improve the capturing performance.

### 5.3.1 Scheduling and Packet Capturing Performance

On multi-core systems, scheduling is an important factor for packet capturing: If several threads, such as kernel threads of the operating system and a multi-threaded capturing application in user space are involved, distributing them among the available cores in a clever way is crucial for the performance.

Obviously, if two processes are scheduled to run on a single core, they have to share the available CPU time. This can lead to shortage of CPU time in one or both of the processes, and results in packet loss if one of them cannot cope with the network speed. Additionally, the processes then do not run simultaneously but alternately. As Schneider et al. [72] already found in their analysis, packet loss occurs if the CPU processing limit is reached. If the kernel capturing and user space analysis are performed on the same CPU, the following effect can be observed: The kernel thread that handles the network card dominates the user space application because it has a higher priority. The application has therefore only little time to process the packets; this leads to the kernel buffers filling up. If the buffers are filled, the kernel thread will take captured packets from the card and will throw them away because there is no more space to store them. Hence, the CPU gets busy capturing

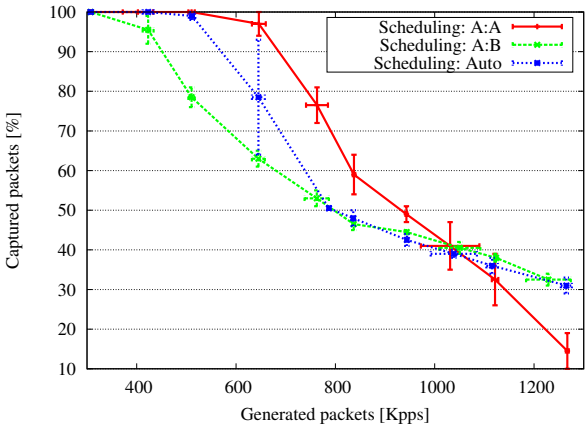



**Figure 5.2:** Scheduling effects

packets that will be immediately thrown away instead of being busy processing the already captured packets, which would empty the buffers.

Conversely, if the processes are scheduled to run on two cores, they have more available CPU power to each one of them and furthermore can truly run in parallel, instead of interleaving the CPU. However, sharing data between two cores or CPUs requires memory synchronization between both of the threads, which can lead to severe performance penalties.

Scheduling can be done in two ways: Processes can either be scheduled dynamically by the operating system's scheduler, or they can be statically pinned to a specific core by the user. Manual pinning involves two necessary operations, as described in [91, 155]:

- Interrupt affinity of the network card interrupts have to be bound to one core.
- The application process must be bound to another core.

We check the influence of automatic vs. manually pinned scheduling in nearly all our experiments.

Figure 5.2 presents a measurement with 64 byte packets with varying numbers of packets per seconds and low application load.

Experiments were run where the kernel and user space application are processed on the same core (A:A), are pinned to run on different cores (A:B), or are scheduled automatically by the operating system's scheduler. Figure 5.2 shows that scheduling both processes on a single CPU results in more captured packets compared to running both processes on different cores, when packet rates are low. This can be explained by the small application and kernel load at these packet rates. Here, the penalties of cache invalidation and synchronization are worse than the penalties of the involved threads being run alternately instead of parallel. With increasing packet rate, the capturing performance in case A:A drops significantly below that of A:B.

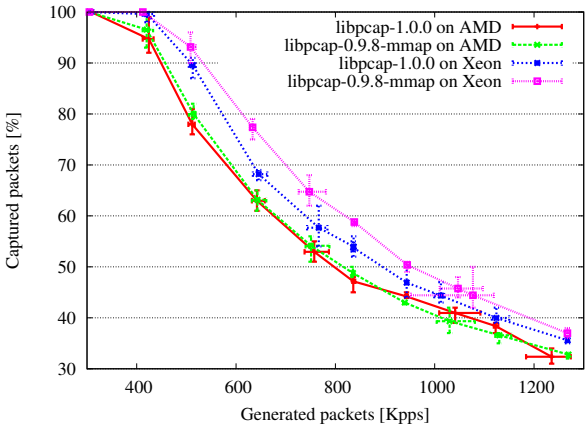



**Figure 5.3:** Comparison of different libpcap versions on Linux

Another interesting observation can be made if automatic scheduling is used. One would expect the scheduler to place a process on the core where it performs best, depending on system load and other factors. However, the scheduler is not informed about the load on the application and is therefore not able to make the right decision. As can be seen on the error bars in Figure 5.2, the decision is not consistent over the repetitions of our experiments in all cases, as the scheduler tries to move the processes to different cores and sometimes sticks with the wrong decision whereas sometimes it makes a good decision.

Real capturing scenarios will almost always have a higher application load than the ones we have shown so far. In our experiments with higher application load which we will show in Section 5.3.3, we can almost always see that running the processes in A:B configuration results in better capturing performance. Static pinning always outperformed automatic scheduling as the schedulers on Linux and FreeBSD almost make wrong decisions very frequently.

### 5.3.2 Comparing Packet Capture Setups under Low Application Load

Applications that capture network traffic usually build on `libpcap` [89] as presented in Section 3.1.1. FreeBSD 8.0 ships libpcap version 1.0.0 in its base system, and most Linux distributions use the same or a more recent version. Not long ago, libpcap version 0.9.8 was the commonly used version on Linux based systems and FreeBSD. As previously discussed, libpcap-0.9.8 or libpcap-0.9.8-mmap were used in most of the earlier evaluations[1].

In this work, we want to use the standard libpcap-1.0.0, which ships with a shared-memory support. As the shared-memory extensions in libpcap 0.9.8 and 1.0.0 where developed by different people, we first want to compare both versions. The results of this comparison are plotted in Figure 5.3 for

1 We refer the reader to Section 3.1.1 for a explanation of the differences between the original library and the MMAP patches.

the AMD and Xeon platforms. We can see that libpcap-0.9.8 performs slightly better on Xeon than libpcap 1.0.0 while 1.0.0 performs better on AMD. However, both differences are rather small, so that we decided to use the now standard 1.0.0 version for our experiments.

Having a closer look at the figure, one can see that the Xeon platform performs better than the AMD platform. This is very surprising as the AMD system's hardware performance is otherwise much faster than the aged Xeon platform. Things get even weirder if we include libpcap-0.9.8 without MMAP into our comparison (not shown in our plot): As the copy operation is way more expensive than the MMAP, one would expect MMAP to perform better than the copy operation. This assumption is true on the Xeon platform. On the AMD platform however, we can observe that libpcap-0.9.8 *without* MMAP performs better than libpcap-0.9.8-mmap or libpcap-1.0.0. This points to some unknown performance problems that prevent the AMD system from showing its superior hardware performance. We will find evidence of this problem throughout our analysis and will identify the cause of the problem later in this chapter.

The next comparison is between standard Linux capturing with PF_PACKET and capturing on Linux using the PF_RING extension from Deri [20]. We use Deri's patches to libpcap-1.0.0, which enables libpcap to read packets from PF_RING. Two important PF_RING parameters can be configured. The first one is the size of the ring buffer, which can be configured in number of packets that can be stored in the ring. Our experiments with different ring sizes reveal that in our setup, the size of the memory-mapped area is not of much influence. We conducted similar experiments with the sizes of Berkeley Packet Filter on FreeBSD and other buffer sizes on Linux. All these experiments showed that increasing the buffer size beyond a certain limit does not boost capturing performance measurably. Instead, we found evidence that too large buffers have a negative impact on capturing performance. These findings are contrary to the findings from Schneider et al. [72], who found large buffers to increase the capturing performance.

The biggest factor regarding the performance is our capturing application—since our hardware, at least the AMD platform, is able to transfer all packets from the network card into memory. If the software is able to consume and process incoming packets faster than wire-speed, the in-kernel buffers will never fill up and there will be no packet loss. However, if the application is not able to process the packets as fast as they come in, increasing the buffer will not help much—rather, it will only reduce the packet loss by the number of elements that can be stored in the buffer, until the buffer is filled.

Schneider et al. sent only 1,000,000 packets per measurement run, whereas we produce packets with 1 GE speed (i.e., more than 100 Megabytes per second), which usually amounts to much more than 1,000,000 packets *per second*, over a time interval of 100 seconds. Increasing kernel buffer sizes to 20 megabytes, as recommended by Schneider et al., allows them to buffer a great share of their total number of packets, but does not help much on the long term. If we increase the buffers to the recommended size, we cannot see any significant improvements in our experiments.

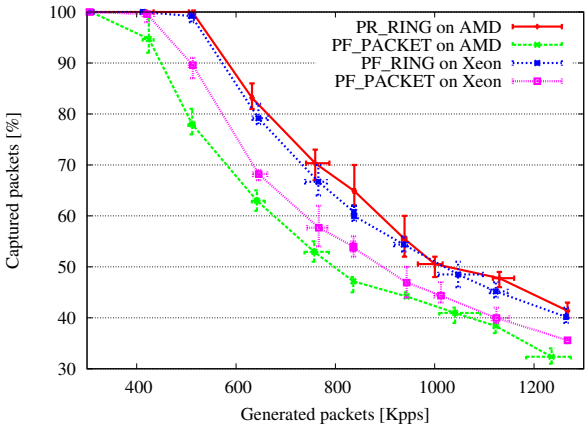



**Figure 5.4:** PF_PACKET vs. PF_RING

Buffer size can be crucial, though: This is the case when the monitoring is not able to process packets at wire-speed, e.g., it can consume up to *N* packets per second (pps), and bursty Internet traffic is captured. If this traffic transports less or equal than *N* pps on average but has bursts with a higher pps rate, then having a sufficient dimensioned buffer to store the burst packets is obviously very important.

The second important parameter is a configuration option called `transparent_mode`. It configures how PF_RING handles packets:

- Transparent mode 0: Captured packets are inserted into the ring via the standard Linux socket API.
- Transparent mode 1: The network card driver inserts the packets directly into the ring (which requires an adopted network driver). Packets are also inserted into the standard Linux network stack.
- Transparent mode 2: Same as mode 1, but received packets are not copied into the standard Linux network stack (for capturing with PF_RING only).

It is obvious that *transparent_mode* 2 performs best as it is optimized for capturing. We conducted some comparisons using different packet sizes and packets rates and indeed found PF_RING to perform best in this mode.

We expected PF_RING to outperform the standard Linux capturing due to the evaluations performed in [20] and [94] when using small packet sizes. This performance benefit should be seen with small packets (64 bytes) and a high number of packets per second, and disappear with bigger packet sizes. Our comparison confirms that PF_RING indeed performs better than PF_PACKET, as can be seen in Figure 5.4. We can also see that PF_RING is better than PF_PACKET on both systems. Furthermore, PF_RING on the faster AMD hardware performs better than PF_RING on Xeon. However, the performance difference is small if one considers the significant differences in

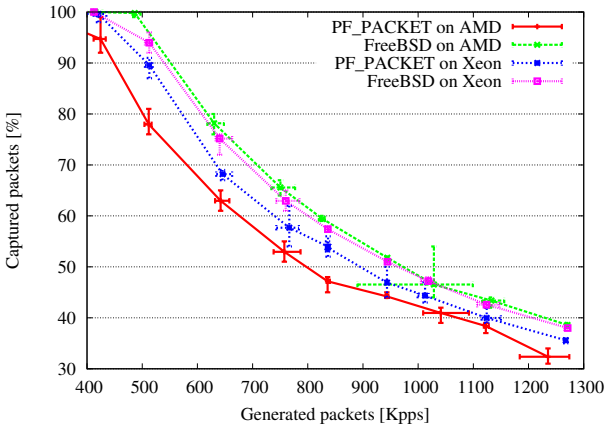



**Figure 5.5:** Linux vs. FreeBSD

hardware, which again points to some performance problems that we pointed out before. One more observation concerns the difference between PF_PACKET and PF_RING within the same platform. Although there is some difference, it is not as pronounced as in previous comparisons. We explain this by the improvements that have been made in the capturing code of PF_PACKET and within libpcap since Deri's and Cascallana's evaluations in 2004 and 2006.

We now compare the measurement results of FreeBSD against the Linux measurements. FreeBSD has two capturing mechanisms: Berkeley Packet Filter (BPF) and Zero Copy Berkeley Packet Filter (ZCBPF) as described in Section 3.1.1. At first, we compared both against each other, but we could not find any significant differences in any of our tests. Schneider et al. found FreeBSD to perform amazingly good even though FreeBSD employed this packet copy operation. This might indicate that the copy operation is indeed not a significant factor that influences the performance of the FreeBSD capturing system. We are uncertain about the true reason for Zero Copy BPF not performing better than BPF; therefore, we do not include ZCBPF into our further comparisons.

Our comparison with the standard BPF is shown in Figure 5.5 and presents the differences between PF_PACKET and FreeBSD. We can see some differences between capturing with PF_PACKET on Linux and capturing with BPF on FreeBSD. FreeBSD performs slightly better on both platforms, which confirms the findings of Schneider et al. [151, 72].

Differences increase if more than one capturing process in running on the systems, as shown in Figure 5.6. This figure shows the capturing performance of FreeBSD, Linux with PF_PACKET and Linux with PF_RING capturing a packet stream of 1270 kpps on the AMD system with one, two and three capturing processes running simultaneously.

Capturing performance on both systems decreases due to the higher CPU load due to the multiple capturing processes. This is an obvious effect and has also been observed in [72]. Linux suffers more from the additional capturing processes compared to FreeBSD, which has also been observed in related work. Amazingly, Linux with PF_RING does not suffer much from these performance

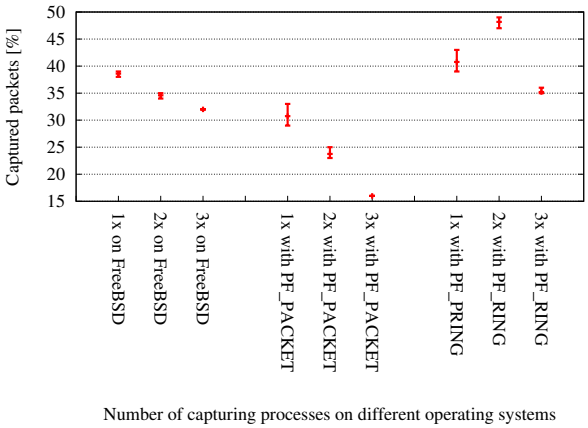



**Figure 5.6:** Capturing with multiple processes

problems and is better than FreeBSD and Linux with PF_PACKET. A strange effect can be seen if two capturing processes are run with PF_PACKET: Although system load increases due to the second capturing process, the overall capturing performances increases. This effect is only visible on the AMD system and not on the Xeon platform and also points to the same strange effect we have seen before and which we will explain in Section 5.3.3.

If we compare our analysis results of the different capturing solutions on FreeBSD and Linux with the results of earlier evaluations, we can see that our results confirm several prior findings: We can reproduce the results of Deri [20] and Cascallana [94] who found PF_RING to perform better than PF_PACKET on Linux. However, we see that the difference between both capturing solutions is not as strong as it used to be in 2004 and 2006 due to general improvements in the standard Linux software stack. Furthermore, we can confirm the findings of Schneider et al. [72] who found that FreeBSD outperforms Linux (with PF_PACKET), especially when more than one capturing process is involved. In contrast, our own analyses reveal that PF_RING on Linux outperforms both PF_PACKET and FreeBSD. PF_RING is even better than FreeBSD when more than one capturing process is deployed, which was the strength of FreeBSD in Schneiders' analysis [72]. We are now comparing the capturing solutions with increased application load and check whether our findings are still valid in such setups.

### 5.3.3 Comparing Packet Capture Setups under High Application Load

So far, we analyzed the capturing performance with very low load on the application by writing captured packets to `/dev/null`. We now increase the application load by performing more computational work for each packet by using the tool *packzip*, which compresses the captured packets using `libz` [154]. The application load can be configured by changing the compression level libz uses. Higher compression levels result in higher application load; however, the load does not increase linearly. This can be seen at the packet drop rates in Figure 5.7.

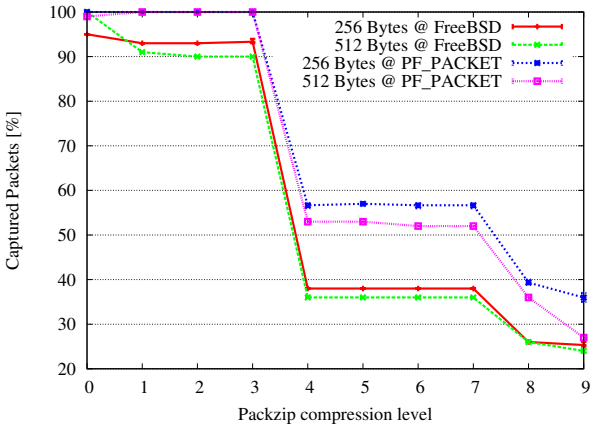



**Figure 5.7:** Packzip on 256 and 512 byte packets

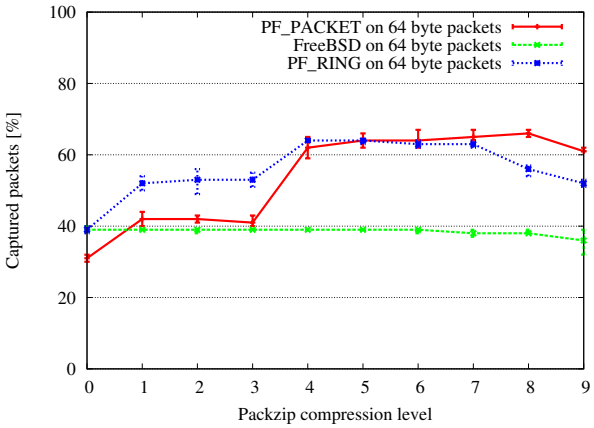



**Figure 5.8:** Packzip on 64 byte packets

The figure presents the results of the capturing process on various systems when capturing a stream consisting of 512 and 256 bytes sized packets at maximum wire-speed at the AMD platform. It can clearly be seen that a higher application load leads to higher packet loss. This happens on both operating system with every capturing solution presented in Section 3.1.1, and is expected due to our findings and the findings in related work. Packet loss kicks in as soon as the available CPU processing power is not sufficient to process all packets. We note that FreeBSD performs worse on our AMD platform compared to Linux with PF_PACKET with higher application load. This observation can be made throughout all our measurements in this section.

However, if we look at a packet stream that consists of 64 byte packets, we can see an amazing effect, shown in Figure 5.8. At first, we see that the overall capturing performance is worse when no application load (compression level 0) compared to the capturing results with 256 and 512 byte packets. This was expected because capturing a large number of (small) packets is more difficult than small number of (big) packets. However, if application load increases, the capturing

performance increases as well. This effect is quite paradox and points to the same strange effect as seen in the experiments before. It can be observed with PF_PACKET and with PF_RING but is limited to Linux. FreeBSD is not affected by this weird effect; instead, capturing performance is nearly constant with increasing application load but almost always below Linux' performance.

In order to better understand what is causing this bewildering phenomenon, we have to take a closer look at the capturing process in Linux. An important task within the capturing chain is the passing of a captured packet from the kernel to the application. This is done via a shared memory area between the kernel and application within libpcap. The shared memory area is associated with a socket, in order to allow signaling between the kernel and the application, if this is desired. We will call this memory area SHM in the remainder of this section.

Packets are sent from kernel to user space using the following algorithm:

- The user application is ready to fetch a new packet.
- It checks SHM for new packets. If a new packet is found, it is processed.
- If no new packet is found, the system call `poll()` is issued in order to wait for new packets.
- The kernel receives a packet and copies it into SHM.
- The kernel "informs" the socket about the available packet; subsequently, `poll()` returns, and the user application will process the packet.

This algorithm is problematic because it involves many systems calls if the application consumes packets very quickly, which has already been found to be problematic in Deri's prior work [20]. A system call is a quite expensive operation as it results in a context switch, cache invalidation, new scheduling of the process, etc. When we increase the compression level, the time spent consuming the packets increases as well, thus less system calls are performed[2]. Obviously, reducing the number of system calls is beneficial to the performance of the packet capture system.

There are several ways to achieve such a reduction. The most obvious solution to this problem is to skip the call to `poll()` and to perform an active wait instead. However, this solution can pose problems to the system: If capturing and analysis are scheduled to run on the same processor (which is not recommended, as we pointed before), polling in a user space process eats valuable CPU time, which the capturing thread within the kernel would need. If capturing and analysis run on different cores, there still are penalties: The kernel and the user space application are trying to access the same memory page, which the kernel wants to write and the user space application wants to read. Hence, both cores or CPUs need to synchronize their memory access, continuously leading to cache invalidations and therefore to bad performance. We patched libpcap to perform the active wait and found some, but only little, improvement due to the reasons discussed above.

2 We confirmed this by measuring the number of calls to `poll()`.

Hence, we searched for ways to reduce the number of calls to `poll()` that do not increase the load on the CPU. Deri already proposed the first one [20]. The basic idea is to perform a sleep operation for several nano seconds if the SHM is found to be empty. Although the sleep operations still implies a system call, it is way better than multiple calls to `poll()`, as the kernel capturing gets some time to copy several packets into the SHM. Deri proposed an adaptive sleep interval that is changed according to the incoming packet rate. We implemented this modification into the libpcap for PF_PACKET. We found that it is not an easy task to choose a proper sleep interval, because sleeping too long will result in filled buffers, whereas a sleep interval that is too short will result in too many system calls and therefore does not solve the problem. Unfortunately, the optimal sleep interval depends on the hardware, the performed analysis and, even worse, on the observed traffic. Hence, a good value has to be found through the end-user by experiments, which requires quite some effort. Deri's user space code that uses PF_RING does not implement his proposed adaptive sleep, probably due to the same reasons. Instead, poll() avoidance is achieved by calls to sched_yield(), which interrupts the application and allows the scheduler to schedule another process. A call to pool() is only performed if several sched_yield() was called for several times and still no packets arrived. Figure 5.8 shows that the algorithm used by PF_RING yield better performance compared to the simple call to poll() with PF_PACKET. However, we can also see negative effects of calling sched_yield() often.

We therefore propose a new third solution that works without the necessity to estimate a timeout and works better than calling sched_yield(): We propose to change the signalling of new incoming packets within `poll()`. As of now, PF_PACKET signals the arrival of every packet into the user space. We recommend to only signal packet arrival if one of the two following conditions are true:

- *N* packets are ready for consuming.
- A timeout of *m* microseconds has elapsed.

Using these conditions, the number of system calls to `poll()` is reduced dramatically. The timeout is only necessary if less than *N* packets arrive within *m*, e.g. if the incoming packet rate is low. In contrast to the sleep timeout discussed before, choosing *m* properly is not necessary to achieve good capturing performance.

We implemented all the solutions into PF_PACKET and libpcap and compared their implications on the performance. For the sleep solutions, we determined a good timeout value by manual experiments. The results are summarized in Figure 5.9. As we can see, the reduced number of system calls yields a large performance boost. We can see that both timeout and our proposed modification to PF_PACKET yield about the same performance boost. Combining both is possible, but does not result in any significant further improvements. We did not have the time to include and evaluate the same measurements with PF_RING, but we are confident that PF_RING will also

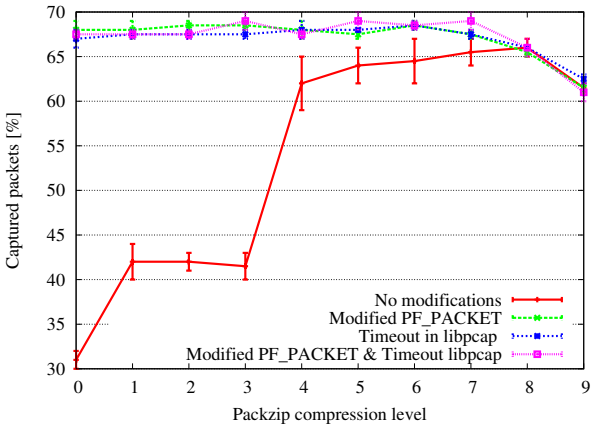



**Figure 5.9:** Capturing 64 byte packets with and without modifications

benefit from our proposed modification. If we look at the CPU utilization, we can see that our proposed PF_PACKET modifications yield in a lower utilized CPU.

### 5.3.4 Driver Improvements

In the experiments presented up to now, we were not able to process small packets at wire-speed, even with our previous improvements. We now move further down the capturing software stack and test driver improvements.

Deri proposed to use modified drivers in order to improve the capturing performance [91]. His driver modifications focus on changing two things previously described:

- Create a dedicated thread for the packet consumption in the driver (respectively for every RX queue of the card).
- Reuse DMA memory pages instead of allocating new pages for the card.

His driver modifications hence help to use the power of multi-core CPUs by spawning a kernel thread that handles the card. This implies that no other thread is scheduled to perform the driver tasks and ensures that there is always a free thread to handle the packets, which is good for the overall performance. Additionally, if interrupts are bound to a given core, the driver thread will run on the same core. It is unclear to us which of the modifications has a bigger influence on the performance. If TNAPI is used with PF_PACKET instead of PF_RING, an additional copy operation is performed to copy the packet from the DMA memory area into a new memory area. The DMA area is reused afterwards.

We tested these improvements with different solutions on Linux but could not consider FreeBSD as there where no modified drivers at the time this evaluation was conducted. Our comparison is plotted in Figure 5.10. In contrast to our previous experiment, we deployed one more traffic generator,

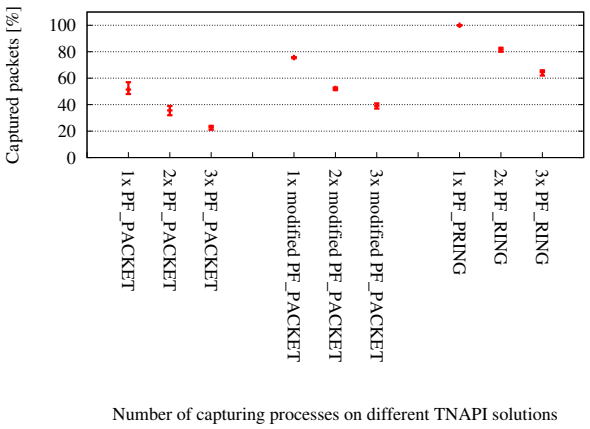



**Figure 5.10:** TNAPI with different capturing solutions on Linux

which allowed us to generate traffic at real wire-speed (i.e., 1.488 million packets). The plot shows results on the AMD system and compares PF_PACKET against our modified PF_PACKET and PF_RING. As can be seen, the TNAPI-aware drivers result in improvements compared to normal drivers. PF_PACKET capturing also benefits from TNAPI, but the improvements are not very significant.

Using TNAPI with our modified PF_PACKET results in good capturing results, which are better than the results with standard drivers and also better than the results with standard FreeBSD. The best performance, however, can be found when TNAPI is combined with PF_RING, resulting in a capturing process that is able to capture 64 byte packets at wire-speed.

## 5.4 Recommendations

This section summarizes our findings and gives recommendations to users as well as system developers. Our recommendations for developers of monitoring applications, drivers and operating systems are listed in subsection 5.4.1. Users of capturing solutions can find advice on how to configure their systems in subsection 5.4.2.

### 5.4.1 For Developers

During our evaluation and comparison, we found some bottlenecks within the software, which can be avoided with careful programming. Developers of monitoring applications, operating systems or drivers should consider these hints in order to increase the performance their systems provide.

Our first advice is targeted at developers of network card drivers. We were able to determine that having a separate kernel thread that is only responsible for the network card can really help to

improve performance. This is especially useful if more than multi-core or multi-CPU systems are available. With current hardware platforms tending to be multi-core systems and the ongoing trend of increasing the number of cores in a system, this feature can be expected to become even more important in the future. In addition, reusing memory pages as DMA areas and working with statically allocated memory blocks should be preferred over using dynamically allocated pages. However, it is unclear to us which of this two recommendations results in the greatest performance boosts.

Signaling between different subsystems, especially if they are driven by another thread, should always be done for accumulated packets. This assumption is valid for all subsystems, ranging from the driver, over the general operating system stacks, up to the user space applications. We therefore recommend the integration of our modifications to PF_PACKET into Linux.

Other areas besides packet capturing may also benefit from our modification: A generic system call that allows to wait for one or more sockets until *N* elements can be read or a timeout is seen, whichever happens first, would be a generalized interface for our modifications to PF_PACKET. Such system calls could be of use in other applications that need to process data on-line as fast as the data arrives.

### 5.4.2 For Users

Configuring a capture system for optimal performance is still challenging task. It is important to choose proper hardware that is capable of capturing the amount of traffic in high-speed networks.

The software that drives the hardware is very important for capture as well, since it highly influences the performance of the capturing process. Our findings conclude that *all* parts of the software stack have great influence on the performance—and that, unfortunately, a user has to check *all* of them in order to debug performance problems.

Performance pitfalls can start at the network card drivers, if interrupt handling does not work as expected, which we found to be true with one of the drivers we used. Checking for an unexpectedly high number of interrupts should one of the first performance debugging steps. Here, enabling polling on the driver could help to solve the issue. However, we found that the POLLING option, a static compile time options for many drivers, did not improve the performance in our tests.

We also recommend using PF_RING with TNAPI, as its performance is superior to the standard capturing stack of FreeBSD or Linux. If using TNAPI or PF_RING is not an option, e.g., because there is no TNAPI-aware driver for the desired network interface card, we recommend using Linux with our modifications to PF_PACKET.

Regardless of the capturing solution used, pinning all the involved threads and processes to different cores is highly recommended in most of the application cases. Using the default scheduler is only recommended when low packet rates are to be captured with low load at the application layer.

Kernel and driver threads should also be pinned to a given core if possible. This can be achieved by explicitly setting the interrupt affinity of a network cards' interrupt.

Finally, if it is not an option to apply one of our proposed techniques for reducing the number of system calls (cf. Sections 5.3.3 and 5.4.1), the user should check if performance improves if he puts a higher load on the application capturing process. As we have seen in 5.3.3, a higher application load can reduce the number of calls to poll() and therefore improve the performance. This holds especially for applications that do not involve much CPU workload, e.g., writing a libpcap stream to disk.

## 5.5 Conclusion

This chapter summarized and compared different capturing solutions on Linux and FreeBSD, including some improvements proposed by researchers. The study was conducted in 2010 with the then-current software. We compared the standard capturing solutions of Linux and FreeBSD to each other, leading to a reappraisal of past work from Schneider et al. [72] with newer versions of the operating systems and capturing software. The evaluation revealed that FreeBSD still outperforms standard Linux PF_PACKET under low application load. FreeBSD is especially good when multiple capturing processes are run.

Our comparison between standard Linux capturing with PF_PACKET and PF_RING confirmed that performance with PF_RING is still better when small packets are to be captured. However, differences are not as big as they where in 2004 or 2006, when the last evaluations were published. Further analyses showed that PF_RING performs better than PF_PACKET if multiple capturing processes are run on the system, and that performance with PF_RING is even better than FreeBSD. During our work, we found a performance bottleneck within the standard Linux capturing facility PF_PACKET and proposed a fix for this problem. Our fix greatly improves the performance of PF_PACKET with small packets. Using our improvements, PF_PACKET performs nearly as good as PF_RING.

Finally, we evaluated Luca Deri's TNAPI driver extension for Linux and found increased performance with all Linux capturing solutions. Best performance can be achieved if TNAPI is combined with PF_RING.

We describe a full capturing setup that follows this guidelines in Chapter 7. The chapter presents an sampling algorithm and discusses how to optimally include the sampling into a multi-core aware capturing architecture. Furthermore, we will provide practical evaluations of the architecture and the sampling on real-world traffic.

# 6 Static Packet Sampling

**Notice of adoption from previous publication:** The text in this chapter contains parts of the paper

- Lothar Braun, Gerhard Münz, and Georg Carle, "Packet Sampling for Worm and Botnet Detection in TCP Connections," in 12th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2010), Osaka, Japan, Apr. 2010. [1]

The text was revised to include minor clarifications. The author of this thesis provided major contributions to the design and implementation of the algorithm. An initial design and implementation of the algorithm was presented by the author as part of his master thesis [156]. This work extends the master thesis by a more thorough evaluation of the algorithms properties as well as its application. The author of this thesis provided major contributions to the evaluation of the algorithm and the interpretation of the results. Additional evaluation of the applicability of the algorithm for botnet monitoring is presented as part of Chapter 8.

## 6.1 Introduction

The previous chapter provided an overview on packet capture system performance and presented improvements to the capturing process. However, packet loss is inevitable under certain circumstances. Application analysis times can be significant depending on the performed analysis. In cases where the per-packet process times exceed the packet-interarrival time, packet loss can occur. Packet loss can have negative influence on the analysis performance of certain analyses.

To keep track of large amounts of traffic, it is possible to increase the processing capacity, for example by deploying specialized signature matching hardware [157, 18] or by distributing packets to multiple systems [74, 75, 158]. The disadvantage of these solutions is that they are costly. As an alternative, we concentrate the available computing power on analyzing the most relevant part of network traffic. For this purpose, we present a new sampling algorithm, which selects packets carrying the first $N$ payload bytes of every TCP connection. We expect these packets to contain sufficient information for various payload-based analysis methods, such as detecting worm and botnet traffic. We have given a motiviation for this assumption as part of Chapter 4.

In order to back the claim of finding relevant parts of the traffic in the beginning of TCP connections, and even UDP biflows, we provide several examples of traffic analysis tasks in Part IV of this thesis. For example, we analyzed a large number of worm and botnet packet traces as a use-case for security-monitoring. The results confirm that the majority of today's worm and botnet traffic can be effectively found by inspecting the early packets of each connection. Of course, this might change in the future, so malicious content could be transmitted after a long series of legitimate data in order to evade detection. We will discuss this problem in Section 6.3.3, and we will present a solution in Chapter 7.

The accurate identification of packets at the beginning of a TCP connection is not trivial if only passive traffic measurement data is available. It requires mechanisms for TCP connection tracking and TCP stream reassembly, which are computationally complex and require a lot of memory to save connection states and to buffer out-of-order packets. Storing the connection states in a hash table may lead to memory exhaustion and loss of connections if the number of simultaneous TCP connections is very high, for example during TCP scans. Furthermore, possible collisions in the hash functions increases the complexity of inserting, querying, and deleting hash table entries. Hardware-based solutions have been presented to cope with these problems [159], yet the implementation of such solutions is very expensive.

We developed and implemented a novel sampling algorithm for deployment in high-speed networks with very high packet rates and large numbers of simultaneous TCP connections. The algorithm selects packets containing the first $N$ payload bytes of a TCP connection by using a simplified TCP connection tracking mechanism and Bloom filters to store the connection states. Compared to existing solutions, the processing complexity per packet is reduced and the required amount of memory is constant at runtime. As a downside, sampling errors may occur because of the simplifications in the connection tracking mechanism as well as due to collisions in the hash functions of the Bloom filters. However, our evaluation of the sampling algorithm shows that high sampling accuracy can be achieved if the size of the Bloom filters is dimensioned appropriately.

We structure this chapter as follows: Section 6.2 presents our packet sampling algorithm, including a description of the deployed simplified TCP connection tracking mechanism and Bloom filter variants. In Section 6.3, we evaluate the probability of sampling errors by applying the sampling algorithm to real traffic traces. Finally, Section 6.4 summarizes the advantages and limitations of the proposed algorithm and gives an outlook on possible extensions.

## 6.2 Connection Based Packet Sampling

The algorithm presented in this section has been first presented in the author's diploma thesis [156]. This section describes the proposed sampling algorithm, which selects those packets carrying the first $N$ bytes of payload of every TCP connection, which requires an appropriate TCP connection

tracking mechanism. Our goal is to provide a sampling algorithm which decides for every single observed packet whether it is picked or dropped. The next subsection sketches the general challenges of TCP connection tracking and discusses the complexity of accurate TCP stream reassembly.

In Section 6.2.2, we present a simplified connection tracking scheme which is more appropriate for connection-based packet sampling in high-speed networks as it maintains much less state per connection. Section 6.2.3 introduces two Bloom filter variants used in our packet sampling algorithm. The implementation details are given in Section 6.2.4, where we will describe how to deploy these Bloom filters and how to use them to store TCP connection states. Finally, we summarize the main properties of the proposed sampling algorithm in Section 6.2.5

### 6.2.1 Accurate TCP Connection Tracking

TCP connections are identified by the addresses and ports of both communication endpoints and the protocol identifier. Start and end of a regular TCP connection are defined by SYN, FIN, and RST packets that are exchanged during the TCP connection establishment and the termination phase. User data is transferred during the communication phase. With help of sequence numbers, the receiving peer is able to detect reordering, loss, and duplication of packets in the network. Packet losses are usually rectified by retransmissions.

TCP connection tracking aims at deducing the connection state as well as the exchanged data by analyzing TCP traffic, which has been passively monitored in the network. A complete analysis of a TCP connection requires to detect the connection establishment and termination, to recognize duplicate packets, and to reorder the packets according to the sequence numbers. Problems may occur if the monitoring process does not observe all packets of the TCP connection. This may happen due to routing changes or asymmetric routing. Additionally, it occurs quite frequently that TCP connections are not shut down properly because of connectivity problems or because one of the communication endpoints disappears without terminating the connection (e.g., due to a system crash). In such situations, an external observer does not know if and when the connection endpoints consider the TCP connection as timed out.

Apart from these general problems, performing accurate TCP connection tracking is quite complex and consumes a certain amount of processing and memory resources for each connection. The accurate reassembly of the exchanged data requires to maintain more or less the same information that is also kept by one of the connection endpoints. In particular, sequence numbers need to be examined to reorder out-of-order packets and to detect duplicates. Furthermore, a buffer is needed to withhold a packet until all preceding packets have arrived. Consequently, accurate connection tracking does not scale well if the number of simultaneous TCP connections is very large, as expected in high-speed networks. Also, we have to cope with exceptional situations resulting in an increased number of TCP connections. Examples are TCP port and network scans as well as SYN

flooding attacks, causing large numbers of so-called half-open connections, which have to be saved by the tracking mechanism until a time-out can be assumed.

If the exact reassembly of TCP connections is not necessary, it is possible to simplify TCP connection tracking in order to improve scalability. We present such a solution in the next subsection.

### 6.2.2 Simplified TCP Connection Tracking

We improve the scalability of TCP connection tracking by reducing the amount of state information that has to be kept per connection as well as the processing complexity per packet. The goal is to keep the decision whether to select or discard a packet as simple as possible while maintaining the sampling accuracy at an acceptable level. Hence, the proposed solution trades sampling accuracy off against scalability and high packet throughput.

The first simplification concerns the detection of connection establishments. Instead of looking for complete TCP three-way handshakes, we consider one SYN packet shortly followed by a second packet without SYN flag as indicator of a successfully established TCP connection. Both packets have to be exchanged between the same endpoints identified by tuples of IP address and port number. We ignore the direction of these two packets, which allows us to detect the beginning of a connection even if we observe only one direction of the traffic, for example due to asymmetric routing. The two packets have to be observed within a small time interval. We determined three seconds to be an appropriate value for this parameter from network traces [156]. Hence, if only a SYN packet is observed, the corresponding state will be automatically deleted after this time-out. There is a small risk of handshake detection errors because we do not check if the sequence and acknowledgement numbers of the two packets are plausible.

The second simplification concerns the connection reassembly: we do not perform any packet reordering nor do we remove duplicated packets. We leave these tasks to the subsequent packet analysis step (e.g., a NIDS) for which knowledge about wrong packet orders and duplicates may even be of interest. Hence, omitting these tasks in the packet sampling algorithm does not necessarily represent a disadvantage. More important is the influence on the sampling results. In order to sample those packets containing the first $N$ bytes of payload, we count the payload lengths in the order of packet arrival without regarding sequence numbers. In the presence of packet reordering and duplicates, we risk selecting packets which should not be sampled (false positives) and risk dropping packets which should be sampled (false negatives). However, we expect that these problems are not very frequent at the beginning of a TCP connection.

Finally, we consider a TCP connection to be terminated if we observe a FIN or RST packet. Again, we do not check if the packet's sequence number is valid, which may cause a problem. We drop all packets observed after the FIN or RST packet. This may result in false negatives if the sampling limit is not reached and if more data is sent from the other peer before shutting down the connection. In order to clean up stale TCP connections which have not been terminated

properly, an supplementary time-out mechanism needs to be implemented which removes the state of connections with long idle times.

As we have seen, the proposed simplifications may lead to wrong sampling decisions under certain conditions. In Section 6.3.2, we evaluate the frequency of false positives and false negatives by comparing the results of accurate and simplified connection tracking applied to real traffic traces.

The simplified connection tracking mechanism still needs to maintain certain state information for every connection. We need to store the arrival of the first SYN packet as well as the counter containing the number of payload bytes to be sampled. The counter must be kept as long as packets are sampled from the corresponding connection, which means until one of the following happens: a FIN or RST packet is observed, the sampled packets reach the configured number of payload bytes, or the connection is timed out.

Although the amount of state data is much smaller than in the case of accurate TCP connection tracking, the required memory still linearly depends on the number of simultaneous TCP connections. As mentioned earlier, the number of TCP connections may grow to very large numbers in high-speed networks or in certain attack situations. To solve this problem, we make use of Bloom filters in which we store the connection states. As a result, the algorithm operates with a fixed amount of memory which is independent of the number of parallel connections. The next section introduces the utilized types of Bloom filters before Section 6.2.4 describes their deployment.

### 6.2.3 Bloom filters

Bloom filters have been introduced in 1970 by Burton H. Bloom [26]. A Bloom filter is a probabilistic data structure that is capable to store information about a set of elements without storing the elements itself. In particular, the stored information specifies whether an element is part of the set or not. The filter is composed of a bit array and a set of hash functions that index the individual bits. Initially, every bit in the array is set to zero, indicating that the set is empty. If a new element is to be inserted into the set, all hash functions must be calculated for this element in order to get all associated indexes. All corresponding bits are then set to one and the new element is considered a part of the set. To check whether an element is part of the set, the associated indexes are calculated in the same way in order to check the bit values. If all bits are set to one, the filter considers the element as part of the set. If at least one of the bits is zero, the element is not part of the set.

False positives are possible due to collisions in the hash functions, meaning that an element can be identified as part of the set even though it has not been added to it. This happens if, and only if, all bits associated to the queried element have been set by other elements. False negatives (i.e., elements which have been added to the set but cannot be identified by the Bloom filter) are not possible.

Memory consumption can be calculated depending on the number of used hash functions ($l$), the collision probability of the hash functions ($p$) and the number of stored elements ($k$) [108]:

$$m = -\frac{l \cdot k}{\ln\left(1 - p^{\frac{1}{l}}\right)} \tag{6.1}$$

A drawback of conventional Bloom filters is that elements cannot be deleted from the set. It is only possible to reset the entire filter, which results in a complete loss of the stored information. For connection tracking, however, we must be able to remove the connection state after the connection terminated. Therefore, we use two Bloom filters variants summarized in the following.

The first variant is called Time-out Bloom filter [109]. Its array is composed of timestamps instead of bits. In order to insert an element into the filter, the associated timestamps are overwritten with the current time. Each element is only valid for a certain amount of time and will automatically expire after this time span. To check if an element is still valid, the difference between the current time and the oldest associated timestamp in the filter is compared to the given timeout.

The second Bloom filter variant is called Count-Min Sketch (CMS) [160] and associates a positive value to each inserted element. Its array consists of counters that can be increased and decreased. With the insertion of an element, a positive value is added to the associated counters. When querying an element, the smallest associated counter contains the current value of the element. If the smallest counter is zero, the element is not considered as part of the set.

We use these two filter variants to store the connection states needed to perform simplified TCP connection tracking and to sample the first $N$ bytes of payload per TCP connection.

### 6.2.4 The Algorithm

We implemented a packet sampling algorithm which selects the first packets of a TCP connection until a maximum of $N$ bytes of payload has been exported. The algorithm uses the simplified TCP connection tracking mechanism presented in Section 6.2.2 and uses two Time-out Bloom filters and one CMS to store the required connection states. We use 2-universal hash functions [161] for the Bloom filters.

Each TCP connection is identified by the quadruple of source IP address ($SA$), destination IP address ($DA$), source port ($SP$), and destination port ($DP$). The element stored in the filters results from the following concatenation:

$$\min\{(SA||SP),(DA||DP)\}\,||\,\max\{(SA||SP),(DA||DP)\}$$

As we sort the tuple of IP address and corresponding port numerically, both directions of the TCP connection map to the same element.

The first Time-out Bloom filter stores the timestamps of all observed SYN packets. The CMS stores the number of payload bytes that need to be exported for an established TCP connection. The second Time-out Bloom filter stores the point in time at which the packet sampling for a given TCP connection was stopped, either because the maximum number of payload bytes was reached or because a FIN or RST packet was observed. The three filters are called *start filter*, *export filter*, and *stop filter* in the remainder of this chapter.

The packet treatment of the sampling algorithm can be summarized as follows:

*On the arrival of a SYN packet*, the timestamp of the packet is written into the start filter and the packet is sampled.

*For any packet which is not a SYN, FIN, or RST packet*, we first check, if a corresponding connection exists in the export filter (non-zero value). If this is the case, the packet is sampled and the counters in the export filter are decreased by the minimum of the payload length and the element's value currently stored in the filter. This is to ensure that the stored value does not become negative.

If the connection does not exist in the export filter, we look up the timestamps stored in the start and stop filters. If the timestamp of the start filter is more recent than the timestamp in the stop filter, and if the timestamp in the start filter is not older than three seconds, the packet belongs to a new connection for which packet sampling should be started. Hence, the packet is sampled and the maximum number of payload bytes $N$ to be sampled minus the payload length of the packet is inserted into the export filter. In any other case, the packet is not sampled.

*On the arrival of a FIN or RST packet*, we check if the connection exists in the export filter. If this is the case, the packet is sampled and the connection is deleted from the export filter by subtracting the current element's value from the associated counters. In addition, the timestamp of the packet is saved in the stop filter. If the connection is not included in the export filter, the FIN or RST packet is not sampled.

As already mentioned, collisions may lead to corrupt information stored in the filters. Consequences are sampling errors in the form of packets which are sampled although they should not (false positives) and packets which are dropped although they should be sampled (false negatives). In Section 6.3.2, we assess the number of sampling errors by applying the algorithm to real traffic traces. Furthermore, we evaluate how many errors are caused by the simplified TCP connection tracking mechanism and how this number increases due to collisions in the hash functions of the Bloom filters.

### 6.2.5 Main Properties of the Sampling Algorithm

Our sampling algorithm approximately samples the first $N$ bytes of payload per TCP connection. Therefore, it implements a simplified TCP connection tracking mechanism which requires less memory per connection state than accurate connection tracking. Since the connection states are

**Table 6.1:** Properties of Network Traces

| Property | Twente | Munich |
|---|---|---|
| Duration (minutes) | 15 | 87 |
| Packets | 16,714,065 | 10,810,511 |
| TCP packets | 11,534,706 | 10,491,400 |
| TCP connections | 35,413 | 18,524 |
| Size | 10.2 GB | 10.9 GB |
| TCP data | 8.9 GB | 10.7 GB |

stored in Bloom filter structures, the memory consumption stays constant during runtime. The computational complexity per packet is constant and does not depend on the number of simultaneous TCP connections or the filling level of the filters. Stale connections may lead to false positives in other connections, yet do not occupy any additional memory. Similarly, TCP scans and SYN flooding attacks fill the start filter and may result in false positives in other connections. However, the required memory remains constant again. In general, the collision probability increases with increasing number of simultaneous connections stored in the filters, which may degrade the sampling accuracy depending on the order of packet arrival.

The *Time Machine* [106] stores a few kilobytes per unidirectional flow, counting packet headers and payload. In contrast, we restrict the total number of payload bytes sampled in both direction. Hence, if the sampling limit is reached, sampling stops for the entire TCP connection, which is useful if both directions are jointly analyzed, such as in an NIDS. Furthermore, counting payload lengths instead of packet lengths is advantageous since empty packets will be selected without decreasing the number of bytes still to be sampled. As the *Time Machine* uses hash tables to store the state of each flow, it requires variable amount of memory and does not scale well if the number of simultaneous flows becomes very high.

The main difference to Canini's approach [118], see Section 3.2.1, is that we count payload bytes instead of packets. Since byte counters are several magnitudes larger than packet counters, Canini's approach cannot be adopted to achieve our sampling goal.

## 6.3 Evaluation

In this section, we evaluate the accuracy of the proposed sampling algorithm by applying it to real traffic traces. Section 6.3.1 gives some details about the traces. In Section 6.3.2, we determine which kind of errors and how many of them actually occur depending on different parameter settings. Finally, we discuss possible ways to evade packet sampling in Section 6.3.3.

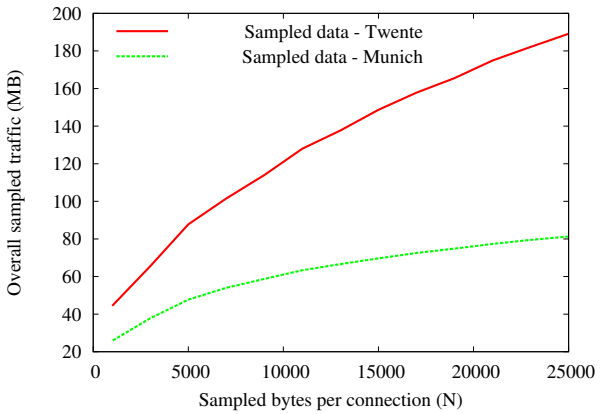



**Figure 6.1:** Amount of sampled traffic

### 6.3.1 Traces Used in Evaluation

We evaluate our algorithms using two traffic traces. The first one was captured at the access link of a student residence at the University of Twente [162]. The second trace file has been recorded in the network of our research group at the Technische Universität München. The two traces will be called *Twente trace* and *Munich trace* in the following. Some properties of the traces are listed in Table 6.1.

We implemented a reference algorithm which provides accurate TCP connection tracking, packet reordering and removal of duplicate packets to identify those packets which should be sampled depending on the sampling limit $N$. Figure 6.1 shows the amount of traffic (including packet headers) sampled by the reference algorithm. As expected, the overall amount of sampled traffic increases for larger values of $N$. However, the amount of sampled traffic is very small compared to the amount of TCP traffic in the original traces, which confirms the observations of previous work [106]. In the case of the Twente trace, between 476,817 ( $N = 1$kB) and 512,334 ($N = 25$kB) packets are sampled. The amount of sampled TCP data is between 44 MB and 189MB, representing 0.5% to 2.1% of the TCP traffic volume. In the case of the Munich trace, the data reduction is even larger: Between 314,063 and 326,742 TCP packets are sampled, corresponding to only 25MB (about 0.2%) to 81MB (about 0.8%) of the TCP traffic. This was expected because the Munich trace contains about half as many TCP connections as the Twente trace at a comparable number of TCP packets. Thus, TCP connections in the Munich trace are longer and contain more packets on average, which increases the effect of the packet sampling.

### 6.3.2 Empirical Analysis of Sampling Errors

In Section 6.2.2, we discussed under which conditions the simplified connection tracking mechanism may cause sampling errors. Additional errors may be caused by the utilization of Bloom filters due to collisions in the hash functions. The goal of our evaluation is to assess how many sampling

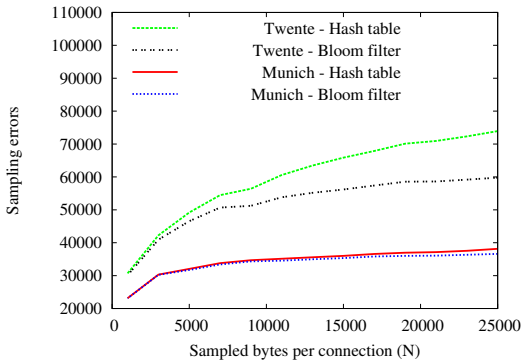



**Figure 6.2:** Influence of sampling limit $N$

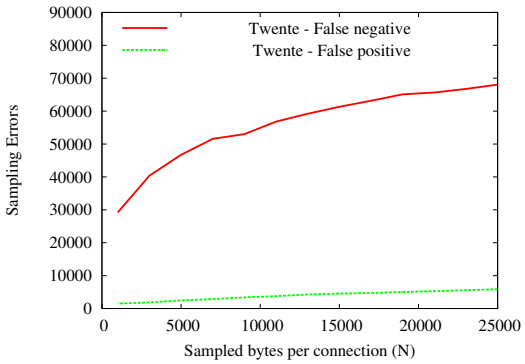



**Figure 6.3:** False positives vs. false negatives

errors are caused by simplified connection tracking and how many errors go back to the Bloom filters. Therefore, we implemented a second variant of the algorithm described in Section 6.2.4 which stores the same connection state information in hash tables with concatenated lists instead of Bloom filters. In various experiments, we determined the numbers of false positives and false negatives for both implementations, taking the output of the reference algorithm as ground truth.

Since the number of hash functions influences the computing cost, a small number of hash functions is desired. On the other hand, a large number of hash functions reduces the probability of collisions. We found that $l = 3$ hash functions is a good trade-off between computational complexity and collision probability [156].

We dimension the Bloom filters for the expected number of connections to be stored simultaneously. In practice, this value needs to be estimated or determined empirically. For our evaluation, we can obtain it with help of the hash table based variant of our sampling algorithm. The maximum number of simultaneous connections that need to be stored in the hash table is 4573 for Twente trace and 1150 for Munich trace. On average, however, only about 1168 and 248 connections are stored, respectively.

Using equation (6.1), we calculate the necessary minimum filter sizes for which the expected collision probability does not exceed 10% ($p = 0.1$). Based on the results, we use filter sizes of 5800 entries for Twente trace and 1200 entries for Munich trace in the following.

We determine the overall number of sampling errors (false positives and false negatives) for both variants of the algorithm. Figure 6.2 shows the results in dependence of the sampling limit $N$. As can be seen, more sampling errors occur in the Twente trace than in the Munich trace, which can be explained by the larger number of TCP connections in the Twente trace. For both variants of the algorithm, the number of errors increases for larger $N$.

In the case of the hash table based algorithm, the errors may only result from simplified TCP connection tracking. With increasing $N$, more packets need to be selected by the sampling algorithm, thus more duplicate packets and wrongly ordered packets can turn into sampling errors. The error

rate is small compared to the overall number of packets in the traces. If 1kB of payload is sampled per connections in the Twente trace, only 30,130 and 30,694 errors are produced by the hash table based variant and the Bloom filter variant of the sampling algorithm, respectively. This corresponds to 7.35% and 7.36% with respect to the number of sampled packets, and to about 0.26% of the total number of 11.5 million TCP packets in the trace. With a sampling limit of $N = 25$kB, the numbers increase to 59,789 (11.6%) and 73,925 (14.4%). Regarding the Munich trace, the difference between the two versions of the algorithm is quite small as well. The hash table based variant causes between 23,106 (6.3%) and 36,601 (11.2%) errors for $N = 1$kB and $N = 25$kB, respectively. With Bloom filters, the figures are 23,131 (6.4%) and 38,156 (11.6%). Apparently, the impact of collisions in the hash functions on the sampling results is smaller than for the Twente trace, which can be explained by different traffic characteristics.

In summary, it can be said that most sampling errors go back to simplified TCP connection tracking. Such errors also occur with other sampling approaches not performing accurate TCP connection tracking, such as the *Time Machine*. The utilization of Bloom filters increases the sampling errors by 0.1% to 24%, depending on the traffic.

For the Twente trace, Figure 6.3 differentiates the sampling errors of the Bloom filter based variant into false negatives and false positives. Most of the errors are false negatives. The likely reason is that packets are retransmitted within the first $N$ bytes of the connection.

Note that we do not count the sampling of a duplicate packet as error because it transports payload within the first $N$ bytes of the connection. However, the missed packets that amount because of the duplicate packets being sampled will be accounted as false negatives.

Now, we evaluate how the number of errors evolves if smaller or larger Bloom filter sizes are chosen. This occurs in situations in which many more parallel connections are observed than expected. Figure 6.4 shows the sampling errors for filter sizes in the range of 2000 to 9000 entries. The sampling limit is set to $N = 5$kB. As expected, smaller filter sizes result in more sampling errors, reaching 76,815 sampling errors for $m = 2000$, which still corresponds to a quite small proportion of 2.24% of all TCP packets. For larger filter sizes ($N > 9000$), the number of errors falls below 47,450 and approaches the number of errors of the hash-based variant of the algorithm (46,548). This means that the impact of the Bloom filters becomes very small.

More important is the observation that the number of errors increases only gradually with decreased filter sizes. Hence, in situations where packets have to be sampled from many more simultaneous connections than originally expected, our sampling algorithm still operates properly, yet at an increased error rate.

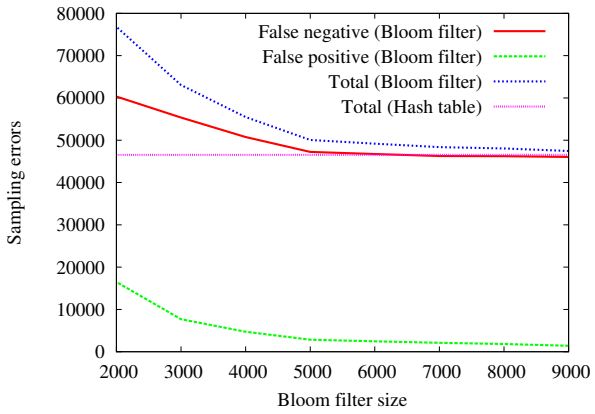



**Figure 6.4:** Influence of Bloom filter size

### 6.3.3 Evading Packet Selection

Evading packet selection, and thus hiding the evaded packets from the analysis, is a potential problem for certain analysis tasks. The most important task related to this problem is security monitoring: an attacker has an interest to hide its attack traffic from an intrusion detection system in order to evade detection and counter measures from the network operators.

An attacker being aware of the sampling strategy could try to evade packet selection and detection in three different ways. Firstly, he could locate the malicious part of payload beyond the first $N$ bytes of a TCP connection. This is a general problem that also affects similar approaches such as the *Time Machine*.

Secondly, he could trick the simplified TCP connection tracking mechanism by inserting packets with identical addresses and ports but invalid sequence numbers as described in Section 6.2.2. Although discarded by the receiver, such packets will be sampled if they are observed within the sampling limit, possibly causing the dropping of later packets with valid sequence numbers that should be sampled. Again, related work such as the *Time Machine* cannot cope with this kind of evasion, either.

The attacker may also attack the sampling algorithm itself, for example by performing a TCP scan in order to poison the values stored in the start filter. This may lead to connections being erroneously added to the export filter causing false positives and an increased sampling rate which risks to overload the analysis system. However, as all SYN packets are sampled, such an attack will be detected. Moreover, the disturbing effect on the sampling result is temporary, which means that the false positive rate will decline to normal level after the scan is over.

Finally, our sampling only selects TCP traffic. If an attacker is able to exploit a security hole by sending traffic in UDP packets, he will not be seen by an intrusion detection system.

To make evasion more difficult, we can dynamically vary $N$ over time to make the sampling limit less predictable. For example, we can choose different values for $N$ depending on some predefined filters, for example in order to sample fewer packets exchanged between trustworthy endpoints. A more sophisticated solution would be to vary $N$ depending on the current load on the detection system. Chapter 7 will present an algorithm that tries to mitigate some of these problems by introducing a dynamic component that varies the sampling limit depending on the resources available for traffic analysis.

## 6.4 Conclusion

We have presented a sampling algorithm that selects packets containing the first $N$ payload bytes of each TCP connection.

The sampling algorithm makes use of a simplified TCP connection tracking mechanism and Bloom filters to store the required connection states. Hence, memory consumption remains constant during runtime, which means that packet losses due to memory exhaustion may not occur. Moreover, the computational complexity per packet is constant and independent of the observed traffic. Thus, the algorithm can be efficiently implemented in software or hardware to sample traffic at high-speed links and pass the selected packets to a subsequent detection system.

An empirical evaluation of the sampling algorithm shows that the number of sampling errors is small and rarely affected by collisions in the Bloom filters if their sizes are appropriately dimensioned for the average number of simultaneous TCP connections. Under extreme conditions with an unexpectedly high number of connections, the number of sampling errors increases slowly without affecting the function of the sampling algorithm.

Although we have only considered TCP connections in this chapter, it would be possible to extend the sampling mechanism for UDP flows, too. In this case, however, we cannot profit from flags indicating the beginning and end of a connection. Chapter 7 will present a sampling algorithm that works agnostic to the used transport protocol and addresses some of the problems discussed in Section 6.3.3.

In Chapter 8, we will provide and analysis of real worm and botnet traffic with *Snort* that evaluates the applicability of our sampling for worm and botnet detection. We will show that the large majority of the signatures are found within the first few kilobytes of payload. Hence, our algorithm can be deployed to reduce the load of the NIDS without degrading the detection results significantly.

# 7 Adaptive Load-Aware Packet Sampling

**Notice of adoption from previous publication:** The text in this chapter contains an extended version of the paper

- Lothar Braun, Cornelius Diekmann, Nils Kammenhuber, Georg Carle, "Adaptive Load-Aware Sampling for Network Monitoring on Multicore Commodity Hardware," in Proceedings of the IEEE/IFIP Networking 2013, New York, NY, May 2013. [6]

The original text from the paper was extended with further information and explanations. A new section on traffic analysis applications and traffic properties was added (Section 7.2). The author of this thesis provided major contributions to the design and implementation of the monitoring architecture and evaluation setup. He developed the concept of using the buffer fill level as an indicator for application performance and network traffic variability, including the idea to adopt the sampling limit based on the observation of the buffer fill level through a sampling process as part of packet capture system. The implementation of the concept was conducted as part of the Interdisciplinary Project (IDP) of Cornelius Diekmann [163] under the supervision of the author of this thesis. The author of this thesis provided major contributions to the evaluation and the interpretation of the analysis results.

## 7.1 Introduction

Section 6.3.3 detailed possible problems with a static sampling approach. One of the major concerns is the requirement for a user to pick an appropriate number of bytes $N$ to sample, as well as the fact that no payload beyond that $N$ will be sampled. In this chapter, we present an adaptive load-aware sampling algorithm that is suitable for security monitoring in single-core and multi-core monitoring environments. Our algorithm adapts the number of packets to be sampled according to the currently observed network traffic and the workload patterns of the analysis processes by adapting the sampling limit dynamically. It aims at fully utilizing the available hardware resources, while at the same time trying to sample those packets that are most likely to contain "interesting" content. Fully utilizing the available hardware requires the exploitation of current multi-core hardware. We therefore also focus on an integrated approach that combines our proposed sampling algorithm with current multi-core aware capturing setups, and discuss how to integrate our work into the systems presented in previous work.

**Table 7.1:** Network Properties

| Network Name | Physical Link Speed | Observed IPs in Monitoring Period |
|---|---|---|
| Network A | 1 GBit | 100,404 |
| Network B | 2 x 10 GBit | 26,394,883 |
| Network C | 2 x 10 GBit | 218,850 |

Another shortcoming discussed in the previous chapter is the limitation of our proposed algorithm to sampling of TCP connections. Certain traffic analyses can benefit from the inclusion of UDP traffic into the analysis, e.g. security monitoring as we will present in Chapter 8. Hence, we aim at an algorithm that can sample from UDP traffic a well.

The remainder of this chapter is structured as follows. Section 7.2 discusses the variability in network traffic over time, and analyzes the behavior of well-known monitoring applications with respect to packet processing times. Section 7.3 introduces the algorithm and describes the capturing architecture that is used to drive the sampling and the analysis process. It covers the problems of capturing and distributing traffic from the network interfaces onto several application instances. Section 7.4 presents the evaluation of our monitoring setup using real-world online traffic from a 10 Gbit/s Internet uplink. We summarize our findings in Section 7.5.

## 7.2 Traffic Volumes and Application Behavior

A dynamic sampling algorithm has to cope with two different variables: the incoming traffic rate and the rate of packet consumption in the application. In this section, we will study traffic volumes and their dynamics to understand how volumes can change. Furthermore, we inspect certain traffic analysis tools and their packet processing times.

### 7.2.1 Measurements of Traffic Volumes

Variability in network traffic can be discussed over long-term and short-term periods. Literature describes variability depending on the time of the day or the week-day [164, 165], or on very short time scales [166]. The variability influences the proper choice of the dynamic sampling limit. It limit is influenced by multiple factors: first, the number of bytes or packets on a link influence, second the distribution of packets and bytes on the flows. In this section, we study traffic over long-term (as in multiple days) and short-term intervals (as within several hours) in different networks to gain a good understanding of the dynamics in these networks.

For our analysis, we have IPFIX or NetFlow data from three networks. We generate time-series information from that data using uniform counter distribution for flows that cross time bucket

**Table 7.2:** Traffic Volumes

| Network Name | Flow Count | Packet Count | Byte Count |
|---|---|---|---|
| Network A | 17,231,625 | 1,558,838,410 | 1.47 TB |
| Network B | 701,980,144 | 549,903,267,678 | 548.94 TB |
| Network C | 2,876,372,134 | 176,895,562,136 | 116.06 TB |

borders as described in RFC 7015 [167]. All three networks have a different size, as in internal hosts, and bandwidth usage on the monitored link.

Table 7.1 lists the relevant properties of the networks. The first network, Network A, is a comparably small network of the networking group at TUM. Network B is a large university network that includes network A. Network C contains data in the data center of a enterprise network. The observation points of Network A and B where located at the upstream links of the respective networks to the Internet. Network C was monitored on two links located near the server farms of the data center. The flow data in Network C contains mostly client requests from remote offices to the different application services that are located on the server farms.

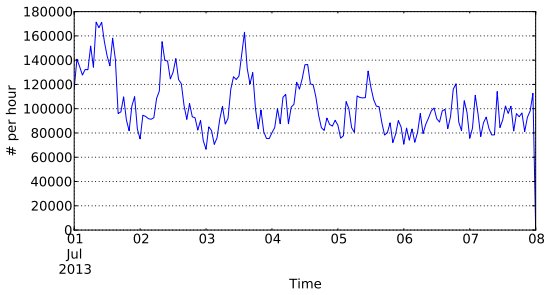
The networks differ in size and purpose. Network A is the smallest network that spans three class C networks. The flow data involves communication between more than 100,000 IP addresses, which have exchanged packets with the systems in the network. Network B is a large scientific network that hosts more than 100,000 devices and spans several class B networks. Packets in this environment have been exchanged between more than 26 million distinct IPs. Flows in Network C involve more than 200,000 devices which acted as servers from within the data centers and the clients in the remote locations.

We collected the flow data over a period of one week in July 2013. The measurement interval spans the days from Monday through Sunday. Table 7.2 shows the overall amount of traffic and flows that have been collected throughout this period.

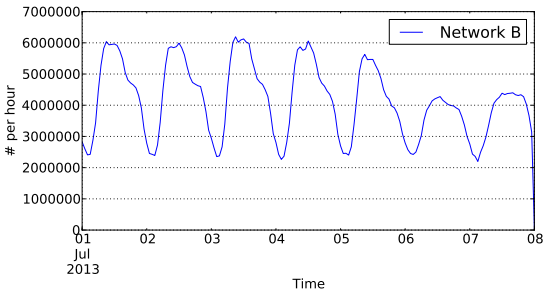
A close look at the numbers provides some interesting insight. The smallest network, as expected, transported the smallest number of flows, packets and bytes over the monitoring period. Over 17 million flows transported roughly 1.5 TB of traffic over the 1 GBit link between the 100,000 IP addresses. Network B, the largest network, transported over 500 TB of traffic in over 700 million flows. When we compare Network C with Network A and B, however, we can make several interesting observations:

At first, the number of total flows exceeds the flow counts in the other networks by large quantities. This finding is not surprising when Network C is compared to Network A: The traffic of Network C involves double as many end systems compared to the small network.

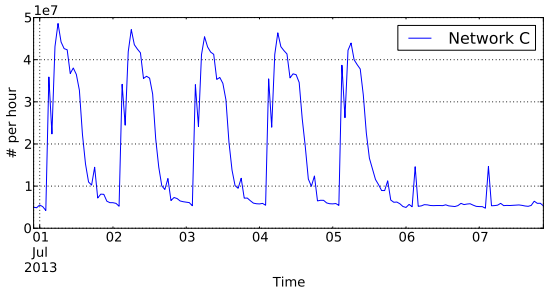
The comparison with Network B is very interesting, however: Network B's flow data involves over two orders of magnitude more IP addresses than that of Network C. Network B exchanges roughly




**(a)** Network A




**(b)** Network B



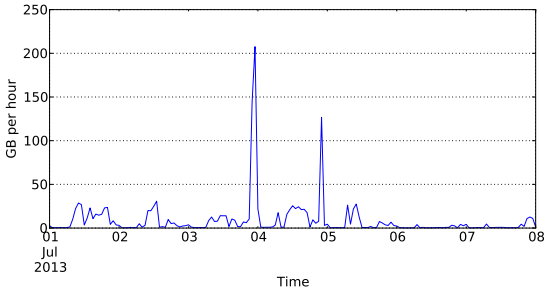

**(c)** Network C

**Figure 7.1:** Number of flows over time

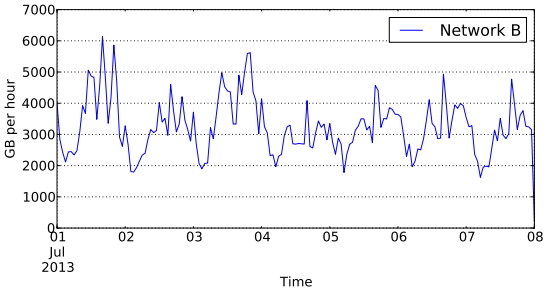
three times more packets and bytes over the monitored links. However, Network C observes around 4 times as many flows than Network B.

Figure 7.1 displays the number of flows over time. We can easily see several trends in the figure, especially in networks B and C. Both have clear indications of day-night patterns as well as weekday and weekend patterns. The weekday and weekend pattern is especially visible in network C: Its main source of traffic is related to business applications, which are limited to the workdays of the week. Network B contains several student dorms and computing clusters which generate a certain amount of traffic through the weekends, resulting in smaller differences between workdays and week ends.

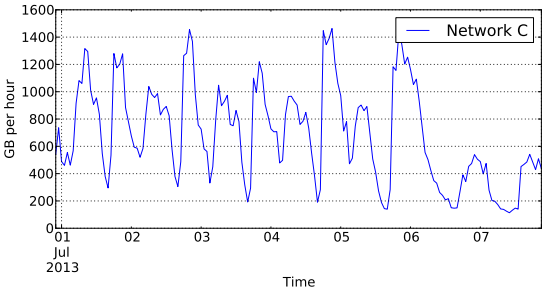
Interestingly, the number of flows in network C exceeds the number of flows in Network B by an order of magnitude. There are two explanations for this finding. The first one is due to the location of the observation points: Network B is monitored at the border gateway router, while Network C is monitored deep within the internal network. Hence, Network C's flows include internal file server, DNS, DHCP and other traffic that is known to have short average flow lengths. The second reason is due to the measurement process. In Network B, the border router generates flow data while in Network C flow data was generated with our software-probe VERMONT [57]. The configuration of the systems differed in active and inactive timeouts. Both were shorter in Network C than in Network B.




**(a)** Network A




**(b)** Network B




**(c)** Network C

**Figure 7.2:** Number of bytes over time

A close look at the byte counters observed over time, which are very important for DPI-based applications, reveals counter values that correlate with the network size. Figure 7.2 provides the details on byte counters on the monitored links. Network B, with many more hosts communicating with each other, produces much more traffic than Network C at all times. Network A, again as expected, exchanged far less traffic over the observation point than both other networks. Each data point in the graphs represents the traffic in a one-hour period. A close look at Network B shows that traffic levels in two following hours can change dramatically. Network A observes even bigger changes, however traffic levels over all are much smaller. Combined with the data provided by the flow counters, we can see that certain variation in the sampling limit is inevitable to provide a constant input to the analysis system in the long run.

The previous statistics can give a feeling for developments in flow and traffic volumes over a long period of time. Short-term bursts are another source of events that can require changes to the sampling limit. We collected a PCAP trace of traffic from one /16 subnet from Network B on Monday, August 26th 2013. The trace spans a time period of two hours and about 96.5 million packets.

Figure 7.3 shows the packet rate in this trace for time bins of one second. We can see that the observed packet rate varies between more than 35,000 and 1,000 packets per second. This packet rate is only a small share of the overall packet rate observed on the link, as will be seen in Section 7.4.

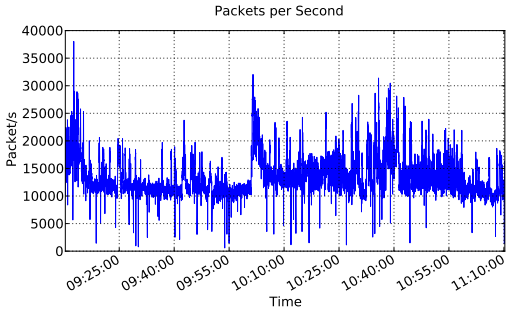



**Figure 7.3:** Packet per second in Network B trace

The changes in the packet rate in two adjacent one-second intervals can vary greatly, pointing to the burstiness of the traffic. A notable increase in the rate can be seen at around 10 a.m. Here, the packet rate increases for several minutes to a higher rate, before dropping eventually. Our sampling algorithm is required to cope with such events in order to avoid overflowing buffers in the traffic analysis application.

Network traffic, however, is not the only factor for traffic analysis performance. Many applications do not perform the same operation on every packet. Instead, the packet content often decides on how the packet is handled. Signature-based regular expression matching in intrusion detection systems is an example for this kind of content-based packet handling: Certain rules can be applied to all packets, others are specific for an application layer protocol and are only applied to packets that belong to the protocol.

### 7.2.2 Packet Processing Time of Analysis Applications

Traffic analysis applications provide numerous different functions for operators and researchers. These functions can range from dumping traffic to disks, providing statistics on traffic volumes, to full-scale payload inspection using regular expression matching. All these tasks pose different loads on CPUs and require different amounts of state to be tracked during the analysis. Hence, they have different requirements for hardware speed and size. We present the traffic analysis performance of different applications that perform packet-level measurements in order to quantify differences between the analysis tasks.

We use the full-packet trace from the previous section to test the packet processing times of several analysis applications. Table 7.3 shows the list of tested applications. The tools perform different tasks, such as network header analysis, packet dumping to disk, or regular expression matching. We chose the applications and tasks according expectation of the performance of the analysis task: *low*, *medium*, and *high* per-packet processing time.

*tcpdump* was chosen for an example of the low-effort category. The application is tested in two different configurations: in the first, it reads the packets and writes them to */dev/null* without further

**Table 7.3:** Traffic Analysis Applications

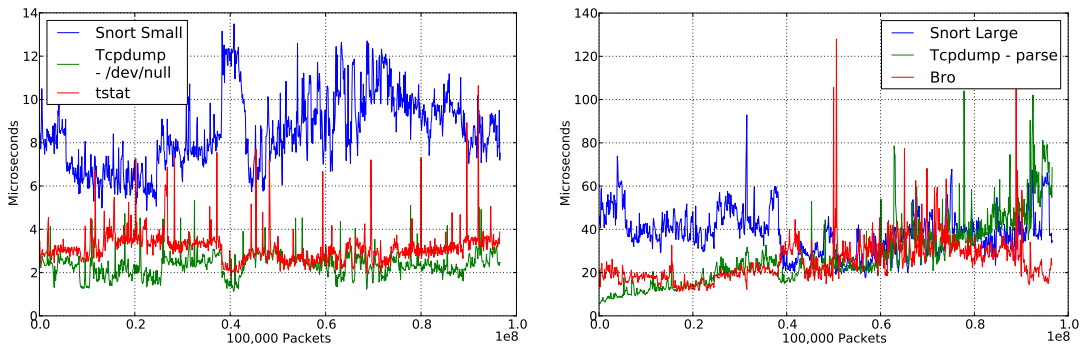
| Application Name | Configuration | Expected Analysis Effort |
|---|---|---|
| tcpdump | Dump to /dev/null | Low |
| tcpdump | Header Analysis | Low |
| Snort | Small rule set | Medium |
| Tstat | Standard configuration | Medium |
| Bro IDS | Standard configuration | High |
| Snort | Large rule set | High |

examination of the content of the packets. This configuration conforms to the testing behavior in Chapter 5, and has been used in several comparisons presented in the analysis of related work in Chapter 3. The second setup adds additional packet handling functions to tcpdump. Instead of writing the packet contents to */dev/null*, the packet content is inspected. The inspection consists of parsing of the network and transport header, as well as some additional calculation for TCP streams. These include the analysis of TCP sequence numbers in order to calculate relative sequence numbers that can lead to a simpler analysis of the sequence number development by a human. The output is written to stdout, which is in this experiment redirected to */dev/null*. Compared to the first configuration of tcpdump, this setup requires the management of certain state within the application, as well as operations on the packet contents.

The *medium*-effort category is filled by two applications: snort and tstat. Snort is a very flexible intrusion detection system, which parses packets and matches captured packets against a configurable rule set. The performance of snort depends on the size and types of the rule sets and the configuration that comes along with them. Our setup configuration contains only very few rules that lack pattern matching but only match on IP addresses. The rule set is described in more detail in Section 7.4.1. Tstat is a TCP statistics tool written in C that examines all packets on the network in order to generate statistics. It provides mechanisms for inspecting network and transport level protocols, as well as support for application identification and can determine and log properties of certain applications such as video downloads or internet telephony. We run it in the standard configuration to output the default set of statistics.

The *high*-effort class also contains two different applications: snort and the Bro IDS. Snort is the same tool as the one used in the medium-effort class, but run with a larger rule set that contains a lot of pattern matching rules. Bro is an intrusion detection and protocol parsing system that consists of a C++-based backend for packet consumption as well as a dynamic scripting language that can be used to implement sophisticated protocol-specific analysis methods [168]. It provides functionality for TCP reassembling [159], application layer payload identification [169], and many application layer analysis modules [170]. We consider the per-packet effort of the system as high due to its rich functionality that is enabled in the default configuration.

In order to measure the per-packet processing times, we added measurement code to the applications. All tools rely on the *libpcap* [89] library for packet acquisition. We modified a copy of libpcap




**(a)** Fast applications **(b)** Slow applications

**Figure 7.4:** Packet consumption timings of different applications

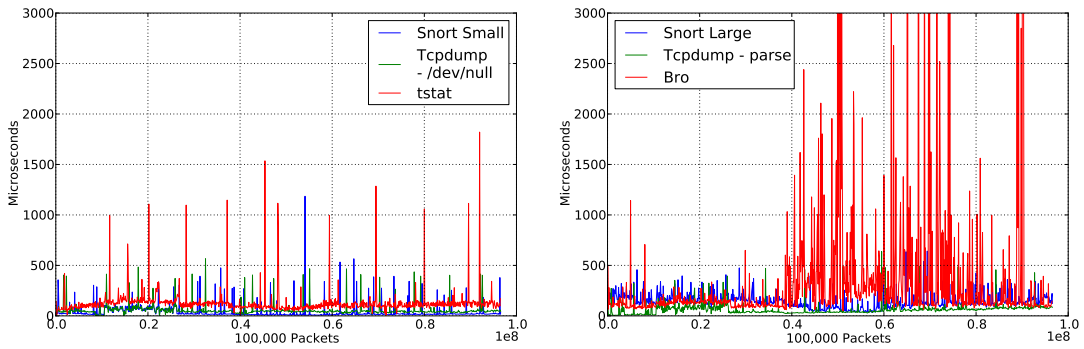
1.4.0 so that it creates timestamps for each packet when they are read from a PCAP file. As the measurement code can add certain additional delay to the per-packet processing times, we tried to minimize the measurement effort, as well as the required state for measurements. In order to achieve this goal, we generated and dumped statistics for batches of packets: Our algorithm calculates the average packet-processing time as well as the variance using well-known online calculation algorithms [171]. We calculate these values for each batch of 100,000 packets and dump the resulting data points to a statistics file.

Figure 7.4 shows the results of the test runs, grouped by the average packet processing times. Applications with an average processing time below 20 microseconds per packet over the complete trace are shown in Figure 7.4a. All others are grouped into Figure 7.4b.

The grouping by results reveals an interesting observation: Tcpdump in packet header parsing configuration is not part of Figure 7.4a, but was assigned to the slow applications. While the processing times at the beginning of the trace are very low, and therefore match our expectations, they increase throughout the processing of the trace. At the end of the trace, they are even higher than those of the other applications. A look at the source code of tcpdump reveals the reason for this: Tcpdump tracks state on TCP connections in order to calculate relative sequence numbers. It uses a very small hash table and uses linear search in case of collisions. Furthermore, tcpdump does not free state about TCP connections once it is established. Over the course of the trace, the hash table fills up and tcpdump spends an increasing amount of time with linear search on the table.

All other applications show packet-processing behavior that is in accordance to the expectation. Tcpdump in the non-parsing configuration shows the smallest workload as it drops all the packets[1]. Tstat requires some more effort to create and maintain the statistics. It also tracks state about the observed connections and flows in order to generate its statistics. However, the state is managed in

1 Please note that although tcpdumps discards the packet, it will nonetheless issue a write system call on a file descriptor that links to /dev/null eventually.




**(a)** Fast applications **(b)** Slow applications

**Figure 7.5:** Standard error of packet consumption times.

a way that does not yield in relevant negative impact to the performance of the system. A likely reason for this is that tstat employs a bigger hash table that yields fewer collisions. For tstat, we can observe spikes in the average processing times.

Snort in its lightweight configuration also provides small processing times lower than 20 microseconds on average. The application requires more time than the previous two applications, and also shows more variability in the processing times. The reason for this is the way Snort decides when to match packets against its rules: in order to minimize the effort of pattern matching, patterns can be restricted to certain ports or application protocols. Snort's per-packet consumption performance therefore highly depends on the incoming traffic, which can vary significantly. Figure 7.4a shows a peak in the processing time for snort around 40 million packets into the trace that lasts for a longer time.

The results for Bro and Snort in Figure 7.4b are only of little surprise. Snort has larger average processing times, which matches the assumption that a larger rule set requires more processing time. Bro has surprisingly good processing times at the beginning of the trace. However average processing times increase throughout the trace, and they become more variable.

This impression increases when we examine the standard error of the average processing time. Figure 7.5 details the standard error for each block of 100,000 packets for all applications. Bro's standard error is shown in Figure 7.5b. We can see that the standard error in the beginning of the trace is almost constant and fairly low. During the second half of the trace, however, we can see that Bro's average packet processing time has a very high standard error. The reader should note that the figure is capped and the actual error values are higher than plotted. It is unclear which effects are responsible the increasing standard error.Increases in the standard errors can also be found within the groups of fast applications in Figure 7.5a. While the increases in the errors are not as heavy as the ones observed for the slower applications, we can however see regular spikes of increased errors. Especially tstat shows periodic increases. As before, the exact reasons for the

spikes are unclear. We can nevertheless observe that different applications tend to experience them throughout the procession of the trace.

### 7.2.3 Implications for the algorithm design

The experiences from our analysis of traffic properties and application packet processing times revealed several important findings. First, we analyzed traffic from three different networks on a larger time scale. We have seen different amounts of traffic in packets, bytes and flows, as well as different relations between these metrics. This can require different sampling limits depending on the network and time of day and day of the week. The algorithm requires some mechanism to cope with long-term changes in the traffic, which can be quite large.

The short-term analysis reveals effects that can lead to dramatic changes in traffic over very short intervals. This burst behavior can make drastic changes within short time intervals necessary to adapt to the changed packet rate. Hence, the algorithm requires mechanisms to adapt to short-term bursts.

Furthermore, our analysis of several applications reveals that packet processing times are not uniform, but can also change over time. The change can be gradual or very bursty with large delays for some applications. The algorithm therefore must not only consider the incoming packet rate, but also the packet consumption rate of the application.

In the following Section, we will present the design of an algorithm that considers both the incoming packet rate and the application behavior. It examines the buffer that is the intersection between the application (consumer) and the network stack (producer) of packets, and adopts the sampling rate in accordance with the fill level of this buffer.

## 7.3 Sampling Architecture and Algorithm

We introduce our sampling algorithm in Section 7.3.1 and discuss in Section 7.3.2 the capturing environment this algorithm can be used in.

### 7.3.1 Adaptive Load-Aware Sampling

A sampling algorithm that selects traffic from the beginning of connections or bi-flows must track state of the observed bi-flows. As discussed in the introduction of this chapter, we want to overcome a short-coming of the algorithm from Chapter 6: we want to get rid of the restriction to sampling of traffic from the beginning of TCP connections and sample from UDP traffic as well. We therefore track communication state based on the definition of bi-flows in RFC 5103 [25], based on the IP-5 tuple in forward and reverse directions. This is compatible with the connection

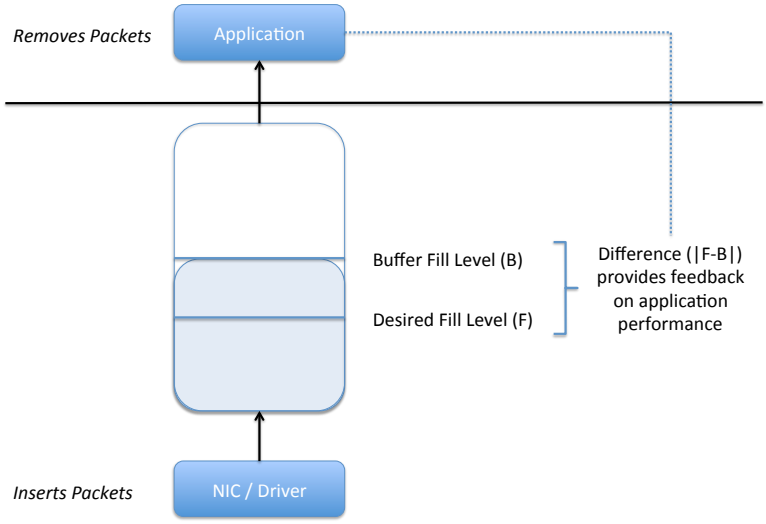



**Figure 7.6:** Buffer fill level feedback mechanism.

identification in Section 6.2.3, with the addition that the transport protocol is included in the key. The implementation of our algorithm uses a fixed-size hash table and is described in detail in the IDP of Cornelius Diekmann [163]. In principle, the algorithm can be used in conjunction with any state tracking mechanism that allows storing and retrieving a current sampling limit with the bi-flow identifiers.

In order to determine a good value for the per-flow sampling limit $N$, we need a notion of how many packets the application(s) can handle. Because we do not want to tailor our sampling algorithm to a specific monitoring application, we cannot make any assumptions about the number of packets that an application can consume. Furthermore, even for specific systems such as Snort, these numbers depend on the system configuration, e. g., detection signatures, and the observed traffic features. We therefore need a feedback system that infers the throughput of the analysis system. The fill-level of the buffer between the capturing and the analysis system can serve as an appropriate performance feedback: Whenever the capturing thread inserts packets at a higher rate than the application can consume, the buffer fill-level will rise. This means that the sampling rate must be decreased by reducing $N$ appropriately. If the application is able to consume packets faster than the incoming rate, the buffer will become nearly empty, and the sampling rate can thus be increased.

When a target buffer fill level $F$ is defined (e. g., one fourth of the overall buffer size), we can measure the deviation $F - B$ of the current fill level $B$ from the target fill level, and use this difference as an indicator for assessing the quality of our current sampling limit. This architecture is illustrated in Figure 7.6. To decide how to adapt the sampling limit $N$, our algorithm can make use of three indicators:

- Current deviation from the target fill level
- Past deviation from the target fill level
- Estimate for the future development of the fill level

We can use a so-called proportional–integral–derivative (PID) controller as a generic feedback controller to model these indicators. In control theory, PID controllers are very popular and have been shown to be the best form of controller if the system to be regulated cannot be modeled more precisely [172]. This allows us to use this system in a setup where we have to cope with unpredictable input patterns, as well as diverse and unknown application behavior.

A PID controller adjusts multiple variables according to dynamic development of several input variables. In our case, only a single variable is observed and used for the adjustment: the deviation $F - B$ of the buffer fill-level from the desired fill-level. Only a single variable is adjusted with the controller: the sampling limit.

In order to adjust the sampling limit, the controller does not only observe the current state of the input variable $P$, but makes use of two more indicators: The integral of the input $I$, and the derivative of the input variable over time $D$. These three terms can be directly mapped to our three previously discussed indicators.

In our setup, the sampling limit $N$ is the variable that is to be manipulated by the PID controller. It is updated at times $t_0, t_1, ...$, which refer to packet arrivals. The sampling limit at time $t_i$ is calculated as

$$n_{t_i} = const + \left(k_\mathrm{P} \cdot P_{t_{i-1}} + k_\mathrm{I} \cdot I_{t_{i-1}} + k_\mathrm{D} \cdot D_{t_{i-1}}\right) \tag{7.1}$$

The proportional term $P$ reflects the current deviation of the buffer fill level $B$ at time $t_i$ from the desired fill level $F$:

$$P_{t_i} = F - B_{t_i} \tag{7.2}$$

The integral term $I$ encodes the past deviation from the target fill level:

$$I_{t_i} = \sum_{j=1}^{i} P_{t_j} \cdot (t_j - t_{j-1}) \tag{7.3}$$

We assume network traffic to be bursty, i.e., quick bursts of packets tend to arrive within very short time intervals. This way, we use $D$ to model extreme changes to the buffer fill level:

$$D_{t_i} = \frac{P_{t_i} - P_{t_{i-1}}}{t_i - t_{i-1}} \tag{7.4}$$

Each of the $P$, $I$ and $D$ terms is weighted by parameters $k_\mathrm{P}, k_\mathrm{I}$, and $k_\mathrm{D}$, which are used to control the influence of the term. Choosing proper weights $k_{(.)}$ for the individual parameters is an important task for tuning the algorithm.

Network environments can have substantial short-lived events such as event-driven packet bursts. Those events, though short-lived, can lead to the buffer temporarily filling up very quickly. Extreme changes in a very short time period bear the potential of sampling limit oscillation and hence unnecessary dropping of packets. In order to mitigate the results of such unforeseeable short-term events, we introduce an additional inertia to the controlling system by applying an exponential moving average (EMA) mechanism. Our new sampling limit does not only depend on the current and past buffer fill level (and its integral and derivatives), but is also influenced by the past sampling limit. We define the final sampling limit $N$ to be constituted from the current PID controller value and the previous sampling limit through

$$N_t = \alpha \cdot n_t + (1-\alpha) \cdot N_{t-1} \tag{7.5}$$

for a user-defined parameter $\alpha \in [0,1] \subset \mathbb{R}$, which weights the current controller-calculated sampling limit against the previous sampling limit.

Finding good parameter sets for the algorithm can have heavy influence on the performance of the algorithm. However, as we show in Section 7.4, a generic parameter set can be found that suits for different monitoring applications under various circumstances. The following heuristics describe the influence of the individual parameters and give hints on how to further tune the parameter sets: The *const* parameter should be set to the desired sampling limit. The $k_\mathrm{P}$ parameter should approximately be set such that a full buffer reduces the sampling limit to zero. Increasing the $k_\mathrm{P}$ parameter strengthens the impact of current fill level deviations from the desired fill level. A too large $k_\mathrm{P}$ parameter can be identified by medium-term oscillations. The $k_\mathrm{I}$ parameter can be used to adapt the long-time sampling limit. Increasing its value gives the controller a larger action scope to automatically find a suitable average sampling limit and to cancel out oscillations induced by the $k_\mathrm{P}$ parameter. However, it decreases the controller's response time that can be induce by packet loss and may lead to long-term oscillations. Finally, the $k_\mathrm{D}$ term increases the controller's response time and acts as counterbalance to the $k_\mathrm{I}$ parameter. A too large $k_\mathrm{D}$ parameter can be identified by short-time oscillations. The parameter $\alpha$ influences the inertia of the system, with a higher value leading to faster changes to the sampling limit.

### 7.3.2 Multi-core Aware Capturing Architecture

For optimally utilizing the available processing power of the hardware, we need a capturing setup that allows to use all resources of the underlying hardware. A capturing setup that optimally utilizes the available processing power must be multi-core aware in order to fully exploit the capabilities of modern commodity hardware.

It implies the need for parallelism in capturing and analyzing software. Since one of our algorithm's goals is to adapt the sampling limit in a way that fully utilizes the available processing power, we discuss how to integrate our algorithm in a multi-core-aware capturing and traffic analysis setup.

Based on our previous work in Chapter 5, we decided to build our setup on PF_RING [20] and TNAPI [91]. However, the algorithm itself is not limited to TNAPI but could also be implemented for other approaches such as PFQ [173] or DNA [174]. We recall from Chapter 3 and Chapter 5: PF_RING provides an optimized capturing module for Linux, which substitutes the standard AF_PACKET capturing module. TNAPI is a driver improvement that creates a kernel thread for the network interface driver. The threads' only responsibility is to move the captured packets into a buffer that is shared between the kernel and the user space.

In combination with Receive Side Scaling (RSS) techniques, where the network card is able to distribute the incoming traffic across multiple CPU cores, multiple TNAPI threads can be used to perform the capturing [91]. Fusco and Deri highlight the importance of proper thread scheduling for the involved capturing and analysis threads [91]. Their recommendation is to create one TNAPI thread and one analysis thread for each available core, and to use the same core for the capturing and the analysis thread in order to allow proper use of CPU caches.

This recommendation substantially involves the network cards capability to load-balance traffic to the RSS queues. Most cards implement per-flow load-balancing which can result in both directions of a connection to be mapped on different cores. However, many network monitoring applications want to observe both sides of the communication. Software-based load-balancing, which is also supported by PF_RING, is therefore required to achieve a proper biflow-aware load-balancing. This bi-flow mapping can result in several flows being re-mapped onto another core, destroying the cache coherency. Our analysis in Chapter 5 revealed that the mapping proposed by Fusco and Deri is good for light-weight analysis processes, but results in higher packet loss for computational expensive setups. As sampling is important in the latter case, we recommend using different cores for capturing threads and analysis processes.

The sampling algorithm can be included into this architecture as part of the analysis process or as part of the packet capture process. This design decision has significant impact on our algorithm: One important benefit of a kernel level implementation is that the filter is executed in the softIRQ context of the kernel. This is beneficial because the softIRQ context is executed independently from the analysis application.

A TNAPI-based setup can ensure that the algorithm is executed for every packet that is inserted into the buffer. It can observe the buffer fill level before a packet is inserted into the buffer, and can therefore adopt the limit very quickly. A user space implementation that runs in the context of the application might not be able to process packets as fast as they are inserted by the kernel, e.g. because the application is blocking. The network stack might therefore insert multiple packets before the user space application can change the sampling limit.

Implementing the algorithm at the kernel level also has one significant drawback: it is way harder to implement a kernel module for the filtering. One reason for this is that floating-point arithmetic cannot be used due to a deactivated floating-point processor [175]. Hence, all calculations must be

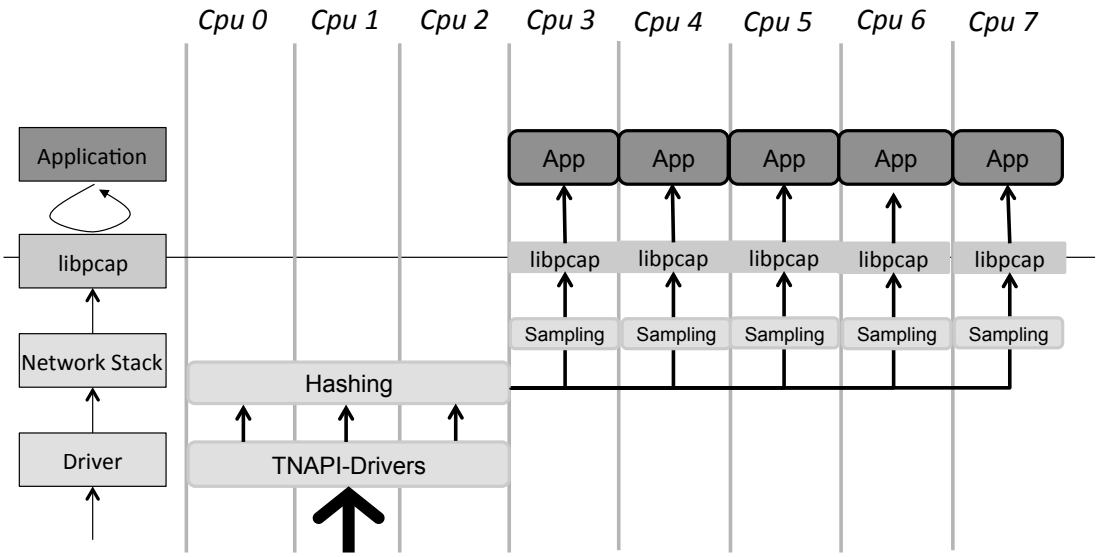



**Figure 7.7:** Data Flow in an Example Setup.

performed using integer arithmetic. This can lead to limitations in the value range of the sampling limit.

For this work, we decided to implement the algorithm as a kernel-level filter that can be included as a plugin into PF_RING: we consider the benefits of fast updates to the sampling limit in cases where the application is working slow more important than a limitation of the sampling limit value range. Fast updates reduce the chances for random packet loss, which is our main goal. It is executed in the context of the kernel and called for every packet that is inserted into buffer.

All these considerations lead to a capturing setup as shown in Figure 7.7. A number of TNAPI threads in the network card driver receive packets from the card, and push the packets to multiple user space analysis applications. TNAPI threads and application threads work independently from each other. Each application has its own ring buffer that is shared between user space and kernel. The TNAPI threads hash the incoming packets and map bi-flows to the user space application buffers.

The application threads read them from these buffers as fast as they can process incoming packets. Our sampling algorithm implementation is included into this setup as a plugin for the PF_RING sampling architecture. It is called by the TNAPI threads before the packet is written to the shared buffer between the kernel and the application. A sampling decision is derived on a per-ring basis: The buffer fill level of each ring is used for the calculation of the sampling limit, resulting in a sampling limit which is adapted for each ring, and hence each application thread. Therefore, thread-specific load characteristics are considered by the sampling process.

## 7.4 Evaluation

Since our algorithm is influenced by packet arrival times and application's processing time per packet, an evaluation should be performed on a real productive system on real network traffic. This is especially important since application processing times depend on the used hardware, monitoring application, and observed traffic features. We therefore start our evaluation with a description of our hardware setup and the network where we were able to deploy our vantage point in Section 7.4.1. The validation of our algorithm in live experiments with different monitoring applications on our 10GE link is described in Section 7.4.2 and Section 7.4.3.

### 7.4.1 Evaluation Setup

A live capturing setup for our evaluation was deployed in the *Munich Scientific Research Network* (Münchner Wissenschaftsnetz, MWN) in Munich, Germany. The research network interconnects three major universities and several affiliated research institutions in the area in and around Munich. Furthermore, the network includes several student dorms that provide housing for the students enrolled in the universities in Munich. Finally, the network hosts a large super-computing cluster that is used by researchers from Munich and other research facilities around the world. In total, the network hosts about 100,000 devices that are used by approximately 120,000 users. It is operated by the *Leibniz Supercomputing Center* (Leibniz-Rechenzentrum, LRZ) and provides Internet access for all its users via a 10 GBit/s link to its upstream provider the *German research network (DFN)*. Our vantage point was deployed on the border gateway between the MWN and its upstream service provider and is therefore able to observe traffic exchanged between the MWN and the Internet. The vantage point is deployed at the routers that generated the traffic for Network B in Section 7.2.

Our monitoring setup was built around standard off-the-shelf PC hardware, operated by a Linux-based operating system. It was bought in 2009 and features a 3.2GHz Intel Core i7 processor with four cores and hyperthreading enabled. Kernel and user space share a total of 12 GB of RAM that have been built into the system. An Intel 10GE network card based on the 82598EB chipset is used in conjunction with a TNAPI driver [91]. Three virtual CPUs are allocated to the TNAPI driver to perform capturing, traffic distribution onto the analyzing processes, and sampling. The other cores are able to run instances of the analyzing application. We implemented our algorithm as a PF_RING filtering plugin, executing in the kernel's softIRQ context of the TNAPI threads.

We employ Snort [21] in different configurations for our evaluation of the algorithm. We show the results for two different configurations with different complexity. These configurations will be called *botcc* and *fullset* in the remainder of this work. The rule sets were obtained on Jan. 08 2013 from emergingthreats.net [131].

The lightweight *botcc* configuration is the free "ETopen" rule set that contains rules for botnet command and control traffic detection. It includes 146 rules with IP addresses of known command

**Table 7.4:** Sampling parameter set for Snort

| Parameter | Value |
|---|---|
| $k_P$ | 3333335 |
| $k_I$ | 0.00093 |
| $k_D$ | 1500 |
| *const* | 1.1 MB |
| $\alpha$ | 0.2 |
| buffer size | 268 MB |

and control servers of different botnets. Snort does not have to perform pattern matching with this configuration but must only check IP addresses of the observed packets against the list of command and control servers.

The second configuration for Snort, *fullset*, employs an emergingthreats.net rule set, too. This rule set, however, contains a total of 11,748 rules. They contain a large number of patterns to match against the observed packets. This configuration is used as a setup to demonstrate the applicability of our algorithm for complex traffic analysis tasks that put high load on the traffic processing engine.

We have seen the influence of the two rule sets in Section 7.2. *botcc* corresponds to the configuration *Snort Small* and *fullset* corresponds to *Snort Large*. The two configurations pose different loads on the system, which allows us to evaluate our algorithm in different scenarios.

We do not only vary the rule sets throughout our evaluation, but also the number of Snort instances that run in parallel. For both rule sets, we run experiments with a single instance and a multi-instance setup. The multi-instance setup consists of four Snort process that run in parallel. Our setup splits the incoming traffic into multiple streams such that each instance has to cope with one fourth of the traffic.

We use a single configuration for our sampling algorithm, as shown in Table 7.4. It shows the PID controller parameters used throughout our experiments, including the multi-instance setups. They were determined in experiments using the heuristics of Section 7.3.1. Our choice of parameters may not be optimal but the results reveal that our solution performs well in different scenarios even with a non-optimal choice of parameters.

### 7.4.2 Simple analysis with lightweight rule set *botcc*

Figure 7.8 compares the number of packets on the link to the number of packets that could be analyzed by Snort with the *botcc* configuration. The figure shows the number of packets per second (pps) that have been captured on the link with the blue line. During the approximately 30 minutes monitoring period, this number of packets varies between 705k and 850k pps, which corresponds

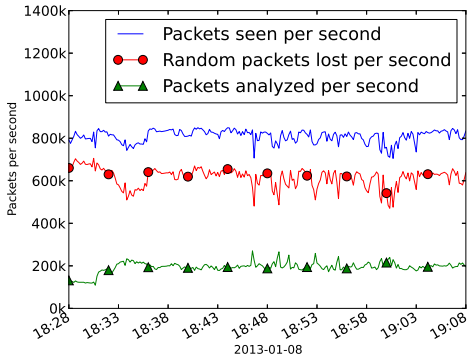



**Figure 7.8:** Snort – Single Instance – Without sampling – *botcc*

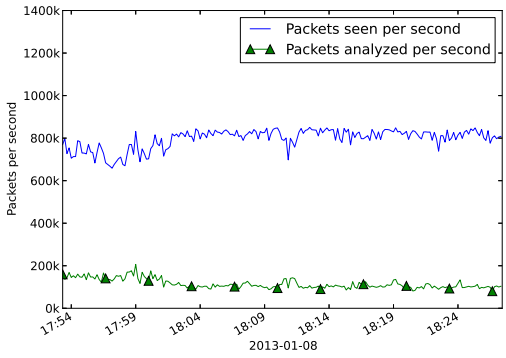



**Figure 7.9:** Snort – Single Instance – Sampling – *botcc*

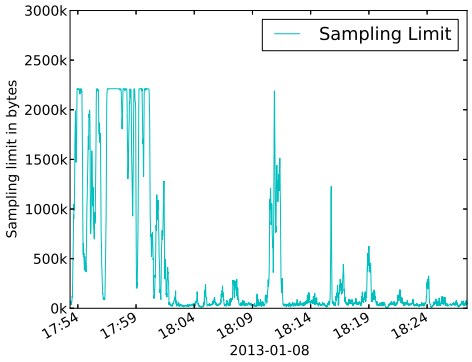



**Figure 7.10:** Sampling limit development – Single Instance – *botcc*

to the maximum number of packets per seconds that are delivered by the border gateway router. On average the incoming packet rate was around 810k pps (mean) or 820k pps (median) during the observation period.

Only a single Snort instance was used to analyze the complete traffic on the link. The number of analyzed packets is far lower as the incoming rate and changes over time depending on the features of the incoming traffic (e.g. number of packets, number of new connections, etc). It varies between about 270k pps (max) and 108k pps (min) at an average of 195k packets per second. Snort's packet processing rate results in a median packet loss rate of about 630k pps. As previous research discussed in detail, e.g. [130, 107], this kind of random packet loss has the potential of losing interesting packets.

We started another monitoring run with the same Snort configuration, but this time we enabled the sampling algorithm. Figure 7.9 plots the packet rate statistics for traffic analysis in this setup.

Incoming packet rates were similar compared to the previous run: 860k pps were observed during peak times, minimum packet rates were around 600k pps with a median of 811k pps.

The number of packets sampled by our algorithm matches the number of packets consumed by Snort. No random packet loss due to full buffers occurred, and all packets picked by the sampling algorithm could be analyzed by Snort. This reveals that our sampling algorithm picks a sampling limit that avoids random packet loss. A closer look at the number of Snort's packet consumption shows a mean number of around 115k pps with a maximum of 213k pps and a minimum of 56k pps. The median packet consumption rate is 105k pps.

One can see that the average packet processing rate in our sampling setup is lower than the processing rate without sampling. The reason for this can be found when taking a closer look at the analysis that Snort performs. The per-packet processing time depends on the characteristics of the incoming traffic. Our sampling algorithm picks packets from the beginning of the connections, which forces Snort to observe all connections on the network. Random packet loss tends to oversample connections with much traffic and tends to miss shorter flows [109]. As Snort keeps per-connection state, this random packet loss decreases the number of internal state Snort needs to manage and thus influences the packet processing time. Those effects are in line with the findings of previous work which analyzed the per-packet processing rates of Snort [176]. The authors find that the packet processing times for packets at the beginning of bi-flows are higher than those packets at later points in a session.

Hence, the lower number of processed packets is an indicator that shows our sampling algorithm works as expected: due to the activated sampling, we observe more connections and thus more payload from the beginning of connections. This increased number in connections leads to a higher packet processing time in Snort. Hence, the Snort instance is consuming packets at a lower rate. Similar effects will be observable in the multi-instance setup and for the *fullset*.

We can observe the behavior of the PID controller when inspecting the development of the sampling limit during this particular monitoring period. Our sampling limit always counts the complete layer four payload including header (TCP/UDP). The first packet that exceeds the sampling limit is passed entirely to the application; only subsequent packets are dropped. Therefore, the first packet of a flow is always sampled. Figure 7.10 plots the sampling limit development throughout the complete monitoring run. The sampling limit starts with our *const* parameter of 1.1 MB sampled traffic per flow, and adapts according to the incoming packet rate and the processing rate of Snort. As the sampling limit is updated on the arrival of a sampled packet, a large number of sampling limit changes can be observed during the monitoring interval. Our kernel module provides average statistics on its sampling limit: Every second a mean value of the sampling limits of the last second is generated.

One can observe heavy changes in the first 10 minutes of the sampling interval, which correlates with the incoming packets rates seen in Figure 7.9. Incoming packet rates in this time interval vary

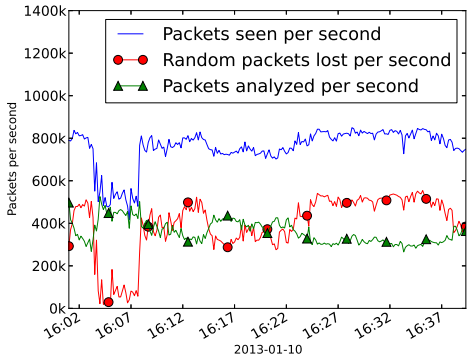



**Figure 7.11:** Snort – 4 instances – Without sampling – *botcc*

highly between 800k pps and 650k pps. The sampling limit adapts to these packet rates: it increases as capturing buffers empty during lower incoming packet rates, and decreases as buffers fill up due to increased incoming packet rates. During this time, the sampling limit reaches a limit set by the implementation and its configuration. As our algorithm is implemented as a kernel-level filter, we cannot use floating-point arithmetic [175] but need to calculate sampling limit updates using integer arithmetic. The maximum sampling limit depends on the parameter set, e.g. buffer size, *const*, and the inertia parameter $k_{(.)}$, $\alpha$. For our configuration, the maximum sampling limit is reached at around 22 MB.

As packet rates grow steadier towards the maximum number of packets the system can consume, changes to the sampling limit become smaller and the sampling limit settles down at much smaller values. However, we can observe increases in the sampling limit as soon as the number of packets on the link decreases. Hence, the system tries to maximize the number of analyzed packets, which is the improvement we desired over the static sampling limit.

In order to study our algorithm in modern multi-core-aware environments, we set up a test run with four instances of the same Snort configuration (*botcc*) running in parallel. This setup increases the performance of the Snort instances, as the traffic is split into four streams. Each instance therefore examines a different part of the traffic, hence, a larger share of the traffic can be analyzed.

At first, we use the configuration with our sampling disabled, and report the performance of the parallel running Snort instances in Figure 7.11. The incoming packet rate, blue line, starts at around 800k pps and remains in this region throughout most of the time. However, at the beginning of the monitoring interval, starting at about 16:05, there is an approximately 5 minute time interval where the incoming packet rate drops to around 500k pps. Unfortunately, we do not have an explanation for the significant drop in incoming packets. However, we can see that packet drops are reduced by a large amount during this time interval.

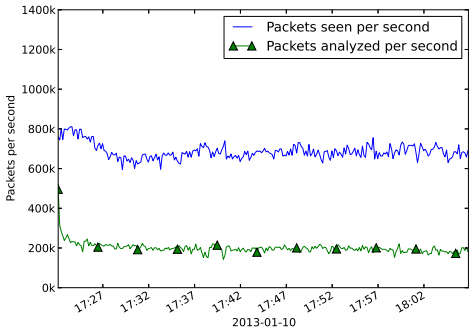



**Figure 7.12:** Snort – with sampling – 4 instances – *botcc*

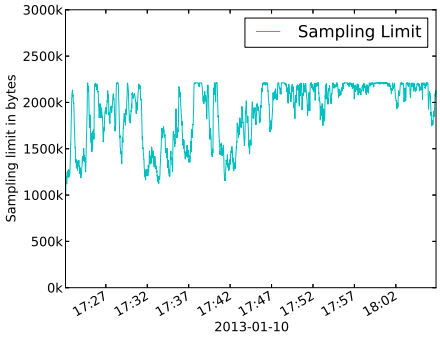



**Figure 7.13:** Sampling limit development – 4 instances – *botcc*

The four Snort instances consume, as expected, a significant higher packet rate, averaging at 360k pps (mean) or 350k pps (median). However, more than half of the packets (396k pps mean / 433k pps median) are dropped due to full buffers. Again, this random packet loss reduces the number of flows that can be observed by the analyzers.

When adding our sampling algorithm to the setup, as shown in Figure 7.12, we can see some changes to the processing rate. While the processing rate in our experiment without activated sampling is larger than 300k pps, we can find smaller average processing times. All Snort instances are able to consume the complete set of sampled packets, at rates that vary between 496k pps and 132k pps with mean and median at about 198k pps. We can furthermore note that the number of incoming packets is smaller than in the run before. It averages at a mean of about 686k pps (median: 678k pps), and varies between 840,000 and 594,000 pps.

There are two reasons for the reduced average packet processing rate. The first one is the same reason as before: Due to the activated sampling process, more connections are observed. This results in an increased per-packet processing time.

The other reason can be found in the sampling limit, which is shown in Figure 7.13. It reveals an increased sampling limit compared to the first monitoring run with a single instance. This is due to the lower incoming packet rate, and the fast packet consumption by Snort. All obtained average sampling limits in our monitoring interval are higher than 10 MB. At the end of our monitoring interval, we can observe that the limit reaches our implementation maximum.

The development of the sampling limit in the beginning of the run suggests that it is close to a good value. However, it seems that it is very close to the maximum in the end of the run. This could be an indicator that good results might also be achieved with parameters that would allow further increase in the sampling limit. Another parameter set could have led to an increased average number of processed packets. However, as we will see in Chapter 8, a sampling limit close to 22 MB is sufficient to retrieve most of alarms generated by Snort.

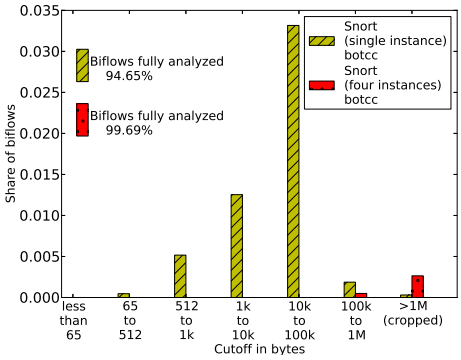



**Figure 7.14:** Biflow cut off – *botcc*

As our plotted sampling limit is a per-second average, the applied sampling limit may vary from the average in certain cases. We therefore cross-check the influence of the sampling limit by examining the biflow cutoff during the setups. A biflow cutoff is the point in the biflow, where the sampling algorithm decides to cut off the connection and stops sampling more traffic from this biflow. This cutoff depends on the current sampling limit when a new packet is observed for a biflow. Whenever the size of the biflow reaches the current sampling limit, no new traffic from this biflow will be sampled, except for the last packet triggering the cutoff.

Figure 7.14 shows the sampling limit cutoff for the previous scenarios for all observed biflows. The single instance setup observed 27.8 million biflows throughout its monitoring period. Out of these, 94.65% were fully analyzed, which means that no sampling cutoff applied for these biflows. The remaining share of the flows where cut off as shown in the figure. The packet for which the cutoff occurs is always sampled, thus, the first packet(s) of each flow are always passed to the application. The class $> 1M$ shows those flows that have not been fully sampled but have had a cutoff that is larger than 1 MB. In the single instance monitoring run, only very few biflows where cut off at 65 to 512 bytes. Most of the biflows that where not fully sampled had a cutoff between 10KB and 100KB. However, even this class represents less than 0.1% of all flows. The four instance Snort monitoring scenario with its larger sampling limit had 99.69% of all flows completely sampled.

### 7.4.3 Complex traffic analysis with rule set *fullset*

In order to determine the results of our algorithm in higher-load scenarios, we performed experiments with the *fullset* rule set. As this rule set includes many more rules than the previous one, we would expect a lower number of packets analyzed by Snort. The results in Figure 7.15 confirm this assumption: A median incoming packet rate of 756k pps translates into a median loss rate of 707k pps.

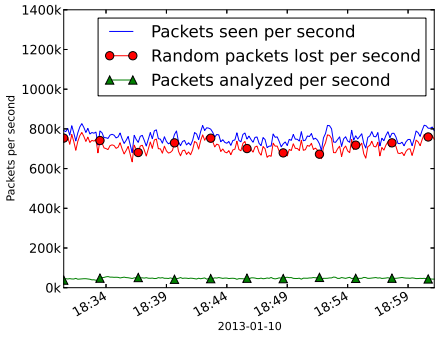



**Figure 7.15:** Snort – Single Instance – Without Sampling – *fullset*

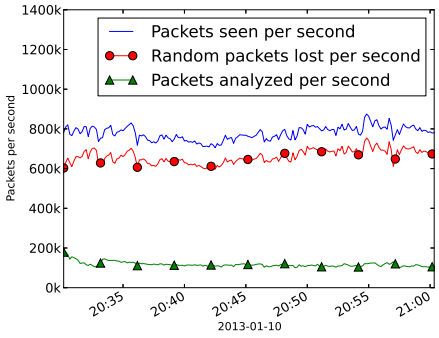



**Figure 7.16:** Snort – Multiple instances – Without Sampling – *fullset*

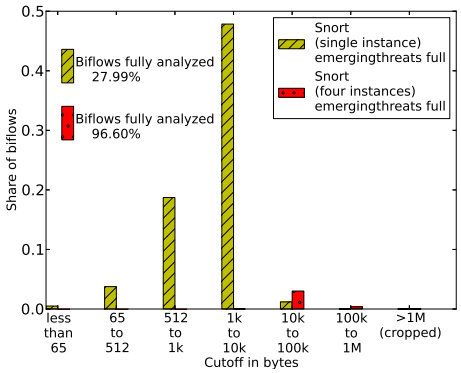



**Figure 7.17:** Biflow cutoff – *fullset*

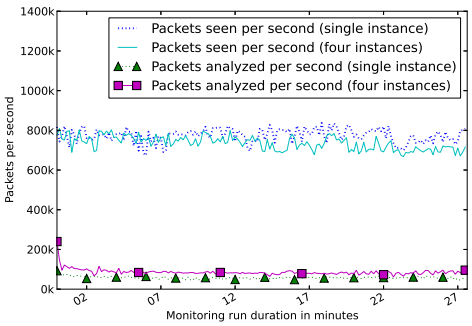



**Figure 7.18:** Snort – With Sampling – *fullset*

This numbers do not increase significantly when four instances are run instead of a single one, as shown in Figure 7.16. This time, an incoming packet rate of 780k pps results in a median loss rate of 659k pps. While adding more analyzing instances increases the analyzed packet rate, the majority of the traffic is still lost due to tail drop on the buffer. We can conclude that the increased number of Snort instances helps. However, we would need a larger number of available cores in order to consume all packets.

When we observe the incoming packet rates for the single and multi instance setups with *fullset* with our sampling enabled, we make similar observations as before. Figure 7.18 shows the figure for the single and multi instance setups. Both have similar incoming packet rates with a median at 781k pps (single instance) and 734k pps (multiple instances), and both setups did not experience any data loss due to full buffers. All packets that have been taken by the sampling algorithm could be analyzed by the Snort instances. The multi instance setup analyzed a median packet rate of 84,000 pps while the single setup only analyzed 58,000 pps. As with the previous rule set, we can observe less analyzed packets when our sampling algorithm is enabled. We suspect the same reasons as before: the sampling algorithm captures more flows which requires the monitoring application to obtain more state.

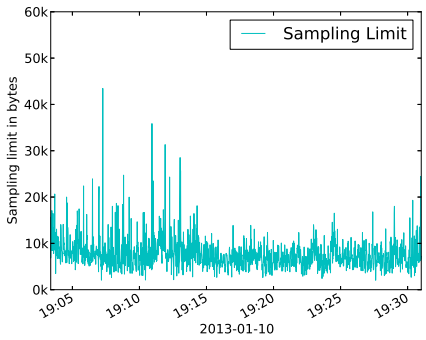



**Figure 7.19:** Snort – Sampling Limit – Single Instance – *fullset*

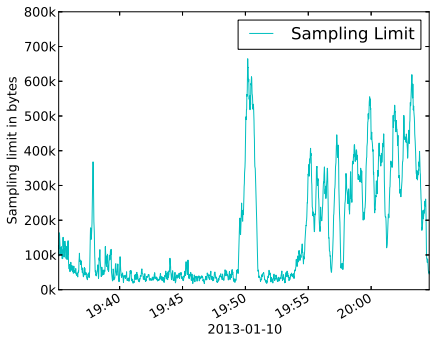



**Figure 7.20:** Snort – Sampling Limit – Multple Instance – *fullset*

A closer look at the sampling limit development reveals bigger differences than the look at the packet consumption rate. Figure 7.19 and Figure 7.20 compare the sampling limit for both setups. While the sampling limit for the single Snort setup can be measured in the region of 10Kb, the four instance setup finds itself with a higher average limit.

The same observation can be made when analyzing the biflow cutoff, which is shown in Figure 7.17 for both setups. The single instance setup, where only one core was used to analyze the complete traffic, was only able to analyzed around 28% of the observed 20,633,534 million biflows completely, and had to cut off the rest of the biflows. Most of the biflows were analyzed to an extend between 1KB to 10KB of their length. The multi instance setup was capable to fully analyze 96.6% of its 20,134,556 million biflows in its monitoring interval. All remaining biflows were cut off somewhere after 100KB with no biflow being cut off with less than 100KB observed.

## 7.5 Discussion

This chapter presented an adaptive load-aware sampling algorithm for high-speed networks. We created an adaptive algorithm that uses a PID controller to adapt a sampling limit with respect to the incoming packet rate and the consumption rate of the analysis application. Our algorithm is a very simple design that only observes a single indicator for both the incoming packet rate and the packet consumption rate: the fill level of the buffer between the network stack and the user space application. A total of three indicators are used to determine a good sampling limit. The past fill level, the current fill level and an estimate for the future fill level. These indicators match perfectly to the input variables of the PID controller and must be weighted against each other for good algorithm results.

We evaluated our algorithm on live traffic on a large university network 10 GE link. The sampling algorithm was configured with a generic parameter set and evaluated in different analysis setups

with differing work-loads. Short-term changes in traffic features can influence the processing rate of monitoring applications such that that they are able to consume more or less traffic than in the previous time interval. Our dynamic sampling algorithm adapts the sampling limit to those short-term events with the result that the available computing resources are fully utilized. It is capable of being used in current multi-core aware traffic setups which run multiple instances of monitoring applications, each on a separate core. These capabilities allow the inclusion of our algorithm into traffic analysis setups that exploit the features of current multi-core hardware.

In the next chapters, we will discuss the applicability of the sampling approach for different analysis tasks. One important task that has been the test-use case for the sampling from traffic of bi-flow beginnings is discussed in Chapter 8.

# Part IV

# Application of Traffic Analysis Systems

# 8 Worm and Botnet Detection

The basic reasoning behind focusing on the packets at the beginning of a flow is that these packets are expected to contain sufficient information to classify the entire flow as harmful or benign. In this section, we evaluate to which extend this assumption is true for worm and botnet traffic.

Signature-based network intrusion detection systems (NIDS) have been in use for a long time. A well-known example, which has already been used in previous chapters of this thesis, is Snort [21]. Snort is a network sniffer that analyzes network traffic in order to detect attacks and other types of undesired traffic. Therefore, packet headers as well as packet payload are checked with pattern matching techniques against a database of attack signatures. Signature-based detection methods are widely used as part of botnet detection mechanisms: Goebel and Holz, for example use regular expression matching for detecting IRC-based botnets in their system Rishi [177]. Gu et al. built BotHunter, a botnet detection system that uses Snort alarms to identify and classify different stages of botnet spread and activity [27]. The tool chain employs Snort with a special botnet-related rule set and employs a correlation engine, which tries to find multiple hints of a botnet infection. Protocol-specific analyses that rely on DPI analysis for botnet detection also aim at finding other types of traffic. Application-specific traffic analysis is used for some important protocols. DNS is an example that is monitored to great extent in various works, .e.g. [178, 179, 180], in order to detect domains that host Command&Control infrastructures to control infected machines. BotHunter also employs DNS-based blacklist detection to identify hosts that resolve malicious domains, or try to connect to IPs that are known to belong to those domains. Other approaches, such as BotMiner, use a combination of payload-based and flow-based inspection [181].

For our evaluation, we focus on signature-based detection strategies. We do this for the following reason: Signature detection becomes more and more complex because the number of attacks to search for increases, and because the signatures themselves become more sophisticated. For example, regular expressions are quite common in today's Snort rules. On the other hand, the amount of packets to be inspected increases with the permanent growth of network traffic. Hence, the resources of a single system are often not sufficient to analyze the entire traffic on a high-speed link. As a result, random packet losses are likely to occur if the traffic exceeds the capacity of the detection system. Signature-based detection mechanisms are therefore a good example for an analysis that can benefit from our sampling algorithms, under the assumption that the relevant attack traffic can be be found in the beginning of TCP connections or UDP biflows.

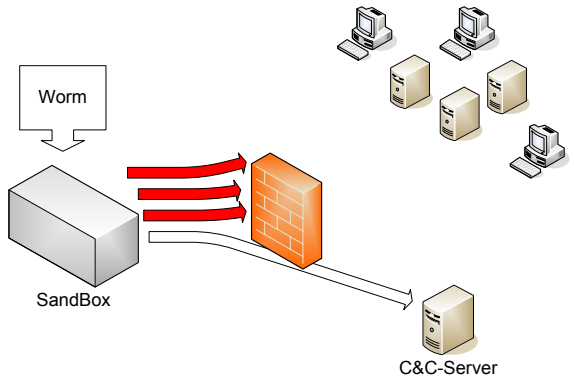



**Figure 8.1:** Dynamic Malware Analysis

We determine the positions within the payload of a TCP connection or UDP biflow at which the matching signatures are found. For this purpose, we modified Snort to output the position of a matching signature in the alarm logfile. As a result, we can evaluate how the detection results are affected if the analysis is restricted to a few kilobytes of payload.

## 8.1 Malware Traces from Dynamic Analysis

**Notice of adoption from previous publication:** The text in this section contains parts of the paper

- Lothar Braun, Gerhard Münz, and Georg Carle, "Packet Sampling for Worm and Botnet Detection in TCP Connections," in 12th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2010), Osaka, Japan, Apr. 2010. [1]

The text was updated to contain more information on the background of dynamic malware analysis and the origin of the traces used in this section. The author of this thesis conducted the experiments and evaluted the results.

In this section, we present an analysis that is based on traces of botnet traffic. Our traces contain the communication of real bots as it could happen in the real world. They were generated in a process that is known as automated dynamic malware analysis [182].

Figure 8.1 schematically describes the process of dynamic malware analysis. In contrast to static analysis, where the binary file is examined using disassemblers and other code analysis tools, dynamic malware analysis relies on executing the worm binary files and logging its interactions with the system and the network. There are multiple ways on how to set up a dynamic malware analysis system. Egele et al. survey a number of available techniques and tools for dynamic analysis [182]. Dynamic malware analysis often employs virtualization or emulation to create an instrumented sandbox that can be used to execute the file. Processes in this virtualized environment

**Table 8.1:** Overview of TCP Alarms

| Alarm | Frequency |
|---|---|
| *Shipped rule set:* | |
| - Shellcode | 45 |
| - Backdoor/Spyware | 38 |
| - HTTP oversize chunk encoding | 12 |
| *Emergingthreats rule set:* | |
| - Instant Messaging | 1116 |
| - HTTP | 810 |
| - Mail | 94 |
| - Scanning | 1041 |
| - Miscellaneous | 355 |

or within the surrounding emulation system log all activities on the system, e.g. system calls of the programs. Furthermore, packet-level measurements capture all interactions of the malware with the network. Many setups, but not all of them, use firewalls to keep certain malicious traffic inside the analysis network. Others use sophisticated tools such as the TrumanBox [183] to redirect malicious traffic to a contained system. Yet other systems do not try to block malicious traffic at all.

Our trace contains traffic from 93 different worms and bots, which have been run in such a controlled environment. The traces have been created by fellow researchers and were provided to us. Every worm sample has been executed several times at different days in order to see different behavior based on the commands received from the botmaster.

The detection is done with two rule sets: the standard rule set shipped with Snort 2.8.4.1 and the rule set downloaded from emergingthreats.net [131] on June 11th 2009. These rules detect a multitude of events ranging from Command&Control (C&C) channels of different bots, shell code, exploits, bot downloads, etc. Altogether, 4030 alarms are generated from the analysis of the traces, among which 3511 alarms are triggered by TCP packets, 451 by UDP packets, and 68 by ICMP packets. Our static Bloom Filter-based algorithm only catches the TCP alarms, as it ignores all other traffic. A version of the algorithm that does not restrict the analysis to TCP could catch the other alarms as well. Our dynamic algorithm is transport layer protocol agnostic and is therefore not affected by this limitation.

Only 95 alarms are raised by the shipped rule set while 3416 alarms go back to the emergingthreats rule set. Table 8.1 shows the alarm frequency for different causes. Alarms described as 'HTTP oversize chunk encoding' are triggered if the chunk length indicated in an HTTP header is larger than the data provided. Most of the emergingthreats alarms are related to scanning activity and C&C channels. C&C channels are realized on top of Instant Messaging protocols, such as IRC, HTTP, or SMTP. Further alarms found in HTTP traffic originate from malware downloads. Finally, a large number of TCP scans are detected, which is a common method for a worm to find new victims. The observed scans are directed to the ports of specific applications, such as VNC (5000-5020),

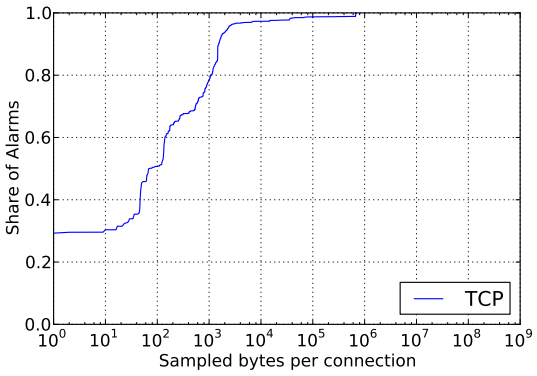



**Figure 8.2:** TCP alarms depending on number of payload bytes

NetBIOS and SMB (135, 139, 445). 'Miscellaneous' contains all remaining alarms which do not fit in any of the other categories.

We calculate the position within the TCP connection at which an alarm is raised using the TCP sequence and acknowledgement numbers as well as TCP payload length of the triggering packet. Hence, the calculated values are not byte accurate but aligned with packet boundaries. This corresponds to the result of our algorithm which samples traffic on a per-packet basis.

Figure 8.2 displays the percentage of TCP alarms as a function of the number of payload bytes $N$ analyzed. As can be seen, most of the alarms are raised very early. About 30% of all alarms are triggered by scans and therefore only need the handshake packets for detection. Most of the alarms (96%) are found within less than 3kB of payload from the beginning of the connections. This means that a sampling limit of a few kilobytes should be sufficient to detect the large majority of today's worm and botnet traffic. The last alarms are found at about 682kB after the beginning of the connection. All of them belong to a generic shellcode rule.

## 8.2 Live Network Traffic Analysis

The previous section presented an analysis of known malicious traffic from malware files. We had a perfect ground truth, as we knew that all triggered alarms where related to malicious traffic. Operators in real world traffic do not have this luxury, but are required to manually analyze the alarms and the packets that triggered the alarms in order to identify which ones are false positives and which originate from real attack traffic. This section of the thesis will analyze real-world traffic to obtain the statistics for traffic for which no ground-truth is available.

Our experiment on live traffic can therefore not reveal how many true positive alarms are lost due to sampling, but how many total alarms are not seen if a certain sampling limit $N$ is applied to the traffic. Our goal is to estimate the quantity of missed alarms that could potentially have interesting information for an operator. Furthermore, researchers presented automatic systems that use Snort

alarms, including the false positives, as input and try to identify the true attacks by correlation of alarms. Raftopoulos and Dimitropoulos, for example, presented an algorithm for automatic alarm mining in large quantities of Snort alarms [184]. They present a heuristic that helps with reducing the number of false positive alarms by examining all alarms generated by Snort. Even missed false positives due to our sampling can have a disturbing influence on this particular algorithm.

Another system that tries to automatically provide reasoning from Snort alarms is the tool *BotHunter*, which has been introduced at the beginning of the chapter. Missing Snort alarms due to a sampling process can also have negative impact on the detection performance of the system. While the work by Raftopoulos and Dimitropoulos is not publicly available, the BotHunter tool can be accessed from the web site [185]. We examine the effect of the missed alarms on the detection rate of BotHunter.

### 8.2.1 Impact of Sampling on Snort Alarm Rates

#### 8.2.1.1 EmergingThreat rules

At first, we evaluate the impact of the proposed sampling on a non-specialized deployment that uses Snort and the EmergingThreats rule set. Our evaluation vantage point is located in the Munich Scientific Network, at the border to the Internet. We use the same setup as described in Section 7.4, in the same network as before. The network is an open university network where people can join with their own devices. In order to find the matching position of the alarms within the observed connections, we need to be able to analyze all traffic on the link.. Our evaluation in Chapter 7 showed that our monitoring setup is not capable to perform this kind of analysis on the link using the full EmergingThreats rule set. For this experiment, we therefore created a smaller rule set which only included a subset of the rules used in the evaluation in Chapter 7. The rule set is composed of the following categories that are defined by the EmergingThreats project:

- attack response
- botcc
- compromised
- exploit
- malware
- scan
- shellcode
- trojan
- user agents

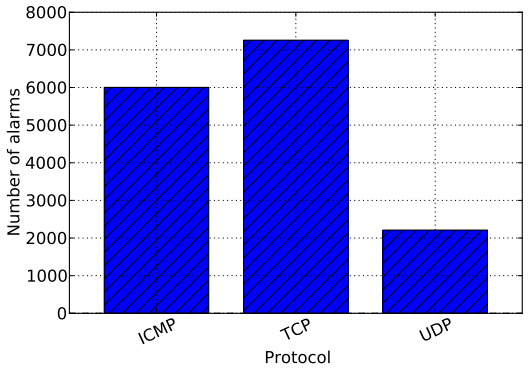



**Figure 8.3:** Number of alarms per protocol

- worms

We picked the rules form these categories in the hope that they contain meaningful rules for detecting botnet related traffic.

This reduction in rules, however, does still not provide sufficient performance gains that allows us to analyze all the traffic on the link. In addition to reducing the number of rules, we also restrict ourselves to traffic from a smaller internal network. The subnet hosts a dynamically allocated address space that is used by client systems.

Our monitoring interval spans roughly two and a half hours on the afternoon of Nov. 4, 2013 using a live capturing analysis using the setup presented in Chapter 7. No sampling was used in order to determine the distribution of alarms within the total traffic using five instances of Snort. Each of the five instances had to process a packet rate of around 15,000 packets per second on average. From this, the five instances of Snort produced a total number of 15,471 alarms. Figure 8.3 plots the alarms grouped by the transport layer protocol. Most of the alarms were generated due to the observation of TCP packets, a total of 7,257 alarms. ICMP traffic was the second largest contributor to the alarms, contributing 6,003. Packets carrying UDP payload only contributed 2,211 alarms, which is the smallest share.

A closer look at the alarms reveals that more than 15,000 alarms were generated by only a small subset of 87 rules. Table 8.2 prints the TOP 10 alarms that have been observed most often. The majority of the alarms have fewer than 100 occurrences in total.

One prominent observation is the alarm with the message *GPL SCAN PING CyberKit 2.2 Windows*. It is responsible for 6,003 alarms and only matches on ICMP packets. Hence, this rule is responsible for all the ICMP alarms in Figure 8.3. The rule matches for a specific payload section in ICMP packets. Every ICMP packet that matches these payload bytes results in an alarm. The ICMP alarms were generated by a total of eight internal hosts that sent the ICMP messages to external IP addresses.

**Table 8.2:** EmergingThreats: TOP 10 Alarms

| Alarm Message | Category | Count |
|---|---|---|
| GPL SHELLCODE x86 inc ebx NOOP | Executable Code was Detected | 6010 |
| GPL SCAN PING CyberKit 2.2 Windows | Misc activity | 6003 |
| ET TROJAN Butterfly/Mariposa Bot client init connection | A Network Trojan was Detected | 885 |
| ET TROJAN Win32.Zbot.chas/Unruy.H Covert DNS CnC Channel TXT Response | A Network Trojan was Detected | 625 |
| ET MALWARE Mozilla User-Agent (Mozilla/5.0) Inbound Likely Fake | A Network Trojan was Detected | 549 |
| ET TROJAN Palevo/BFBot/Mariposa client join attempt | A Network Trojan was Detected | 214 |
| ET MALWARE Simbar Spyware User-Agent Detected | Potential Corporate Privacy Violation | 181 |
| GPL SHELLCODE x86 NOOP | Executable Code was Detected | 131 |
| ET MALWARE MarketScore.com Spyware Proxied Traffic | Potential Corporate Privacy Violation | 82 |
| ET COMPROMISED Known Compromised or Hostile Host Traffic UDP (18) | Misc Attack | 79 |

The most often seen alarm for TCP is *GPL SHELLCODE x86 inc ebx NOOP*, and is classified as *Executable Code was Detected*. An alarm is generated whenever the content *CCCCCCCCCCC-CCCCCCCCCCCCC* is detected on certain ports. The character C translates to the x86 NOOP operation and can be used as part of SHELLCODE. A threshold of 24 consecutive NOOPs lead to the alarm. Unfortunately, such generic rules are known to produce a high false positive rate when binary data is transmitted in a network. We examine the involved ports and servers, and find that most of the alarms where generated on HTTP traffic from high-profile web sites. As these alarms are easily triggered by binary data and occur with a very high rate, they are unlikely to be triggered by real malicious traffic. Manual inspection of the alarms is unfeasible due to the high number of events. Our conclusion is that most of the alarms are due to false positives and that they are of little interest to administrators. We therefore remove them from our alarm body for determining the position of alarms within the stream sizes.

The other alarms seem to have at least some relevance to operators, such that operators might be interested in seeing them. If sampling is applied with a certain threshold, these alarms might get dropped due to the fact that they are located in the transport stream outside the sampling limit.

For TCP connections we calculate the position at which an alarm is raised using the TCP sequence and acknowledgement numbers as well as TCP payload length of the triggering packet. The calculation matches the one performed in Section 8.1. It is not possible to perform the same calculation for UDP due to the lack of this information in the UDP headers. We use Snort's definition of a UDP session as a help for determining the position of the payload. Snort considers all packets being exchanged between two UDP end points, identified by IP addresses and ports, within a certain period of activity to belong to a single session. This matches the definition of a bi-flow as per RFC 5103 [25] with only an inactive timeout applied to the flows. Our inactive

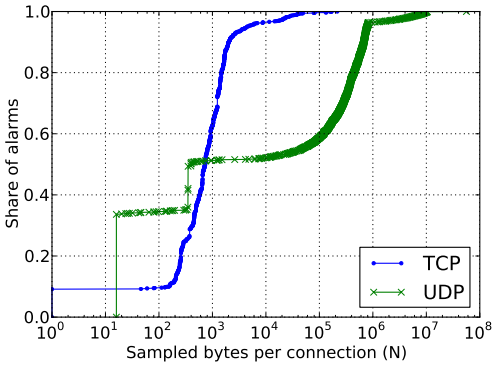



**Figure 8.4:** Alarms depending on the number of payload bytes

timeout was configured to be 180 seconds. We calculate the position within this UDP streams by accumulating all payload bytes of the packets exchanged within the session.

Figure 8.4 plots the cumulative distribution of the alarms for the two protocols. The figure reveals how many alarms would have been generated if a certain sampling limit for our algorithm was applied for both UDP and TCP.

TCP alarms on live traffic have a similar distribution as has been observed before on the malware traces in Section 8.1. A certain amount of alarms are triggered without any payload being sent. The malware traces had 30% of alarms due to scanning behavior where alarms could be found without payload. Our live traffic alarms contain only 10% of alarms that do not require payload. These are only to a small amount part of scanning activity. Others are triggered because of connections to blacklisted IP addresses which are shipped by EmergingThreats.

Most alarms that trigger on payload match for payloads at a position between 100 bytes and 10 kB. 60% of all TCP alarms are triggered from application payload that resides in the first kilobyte of a connection. Almost all alarms can be found if 100 kB of the exchanged payload is observed. This finding also matches the findings of alarm positions obtained in the previous section from malware traces.

The distribution of UDP alarm position differs from the distribution of TCP alarms. At first, not a single alarm is found that does not require payload. This finding is of little surprise since UDP does not have any control packets that do not contain payload. So even if an alarm is found on the first packet, this packet is expected to carry payload and therefore contributes to the stream length. Over 30% of the alarms do not require more than 300 bytes of payload for a match. Due to the little amount of traffic, it is very likely that the match occurs on the first packet that is exchanged within a session. Over 50% of all alarms can be found within the first 10 kB of exchanged payload. The increase between 30% and 50% at a stream size between 300 and 400 bytes is caused by a single rule which matches to messages which have mostly identical sizes. For TCP we found that

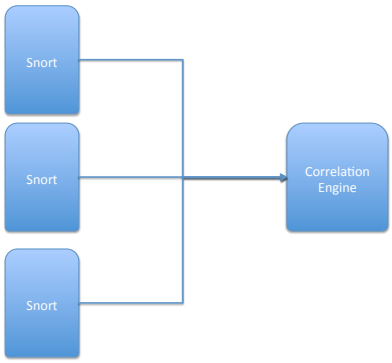



**Figure 8.5:** Interaction between Snort and Correlator

10 kB of traffic are sufficient to capture most of the alarms. In order to obtain a similar share of the UDP alarms, we must sample up to one megabyte of payload of the communication.

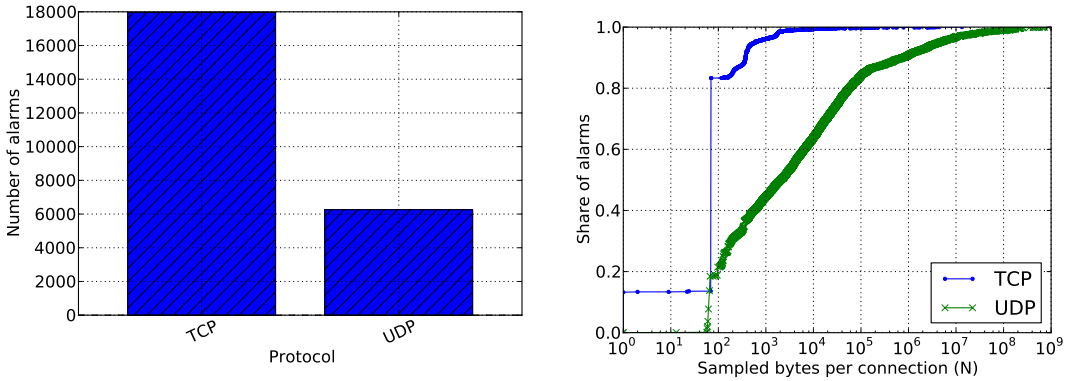
#### 8.2.1.2 BotHunter rule set

Due to the high number of false positive alarms that are triggered by benign traffic, our impression on the impact of the true positive alarms is not very accurate. We therefore rely on previous work to get a better understanding these alarms by using an existing tool that has been developed for botnet detection: BotHunter [27]. It consists of two components as shown in Figure 8.5. The first is a tuned and enhanced version of Snort. The enhancements include new detection modules, including DNS message analysis for blacklisting and other modules. Furthermore, BotHunter comes with a rule set that is claimed to be a useful selection of detection rules for traffic that is related to botnet traffic. They are chosen from multiple rule set based on a manual selection process by the authors of BotHunter.

Snort is a GPL-based open source tool, which means that all changes made to Snort are available as source code. The correlation engine that consumes the output of Snort is closed-sourced and only available as pre-compiled binary. It consumes the alarms generated by Snort and provides correlated results.

Most important for our evaluation is the fact that the design of the system allows us to apply our patches to the modified version of Snort. Snort can be run in our setup to generate alarms using online monitoring. The alarms can be fed into the correlation engine afterwards in an offline mode.

Our online monitoring setup for the BotHunter version of Snort equals the previous setup. We deployed the configuration on the same network that was used in Section 8.2.1.1 in the evening hours of Nov. 4th, 2013. Alarms were collected for a time period of roughly four and a half hours. This measurement run resulted in a total of 24,223 alarms.




**(a)** Number of alarms per protocol

**(b)** Alarms depending on the number of payload bytes

**Figure 8.6:** Alarms for the BotHunter rule set

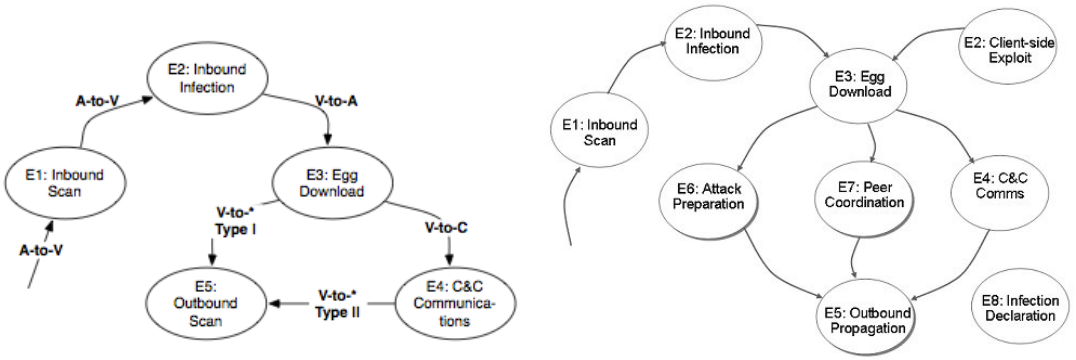
Figure 8.6a shows the distribution of these alarms among the transport layer protocols. Interestingly, not a single ICMP alarm was generated during the time period. The reason for this lack of ICMP-related events is the BotHunter configuration: Its rule set only contains two rules that match on ICMP traffic. Both of them are very specific and search for uses of Command & Control Messages of a botnet which uses ICMP messages to distribute commands.

More alarms are triggered due to TCP traffic than for UDP traffic (17,966 vs. 6,257). A closer look at the distribution of alarm positions within the communication session in Figure 8.6 reveals that many of them are very early within the sessions. Less than 20% of TCP alarms originate from connections that did not exchange any traffic prior to the alarm. This is around the order of magnitude that we have seen in the other experiments on live traffic and malware traffic.

Most of the TCP alarms occur within the first 70 payload bytes of the TCP connections. This is due to a single rule that matches for BitTorrent traffic, which is in use in the network. Normally, we would consider this a clear false positive and drop these alarms.. However, we do not remove the alarms from the set because the BotHunter correlation engine is expected to make sense of such alarms.

Despite this single rule exception, the remainder of the distribution of the TCP alarm positions follows the same rules as the ones seen before: 10 kB of payload from a connection are sufficient to retrieve most of the alarms. This shows that our sampling algorithm is suitable for multiple different rule sets.

A similar observation can be made for UDP traffic: only little payload from UDP sessions is sufficient to retrieve most of the alarms. It is necessary to sample more of the UDP payload compared to TCP in order to obtain a comparable share of alarms. This result is in line with the findings of the previous experiment.




**(a)** Original Model [27] **(b)** New Model [185]

**Figure 8.7:** BotHunter Dialog Models

### 8.2.2 Impact of Sampling on BotHunter Detection Rates

BotHunter uses a dialog model to determine infections, which aims at reducing the false positive rate of Snort alarms. The dialog models the typical communication of a bot life, such as scanning, infection, egg download, C&C traffic and outbound scanning. In order to identify an infected system, BotHunter tries to find multiple signs of botnet communications, e.g. alarms from multiple communication categories. The work by Gu et al. describes the initial correlation engine and the processes that are applied to the incoming snort alarms [27]. BotHunter's original detection model as defined in the publication is shown in Figure 8.7a.

Since the original publication in 2008, several improvements and new releases have been issued. BotHunter's authors extended the initial functionality of the system, including an enhanced Dialog Model which uses more activity categories. The extended dialog model that is used by our version of the engine and is shown in Figure 8.7b.

Each rule must be supplied with a classification that matches one of the life-cycle elements in the dialog model. The rules that are shipped with BotHunter contain the necessary classification. Alarms are consumed by the correlation engine and a number of checks are applied in order to decide which of the alarms point to a real infection. The correlation engine itself is closed-source, which means it is not possible to reconstruct the exact matching algorithm.

When BotHunter finds an alarm, then it presents a so-called profile. A profile lists the infected IP address as well as the evens that caused BotHunter to raise an infection alarm. It is possible that a single IP address can have multiple different profiles.

Our first experiment with the correlation engine involves all alarms that were generated by Snort during our monitoring runs. These are the alarms that have been previously presented in Section 8.2.1.2. BotHunter produces a total of 29 profiles when all alarms are analyzed by the correlation engine. The 29 profiles involve 8 IPs which have been flagged as infected systems. Figure 8.8 plots the

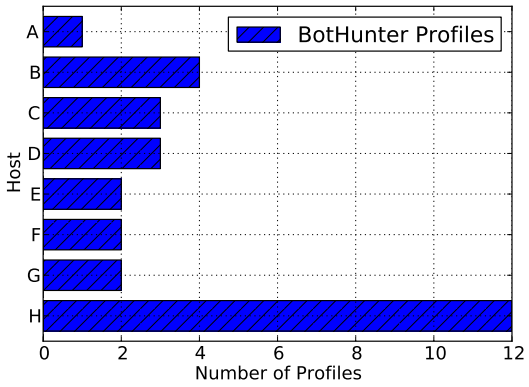



**Figure 8.8:** BotHunter Alarms

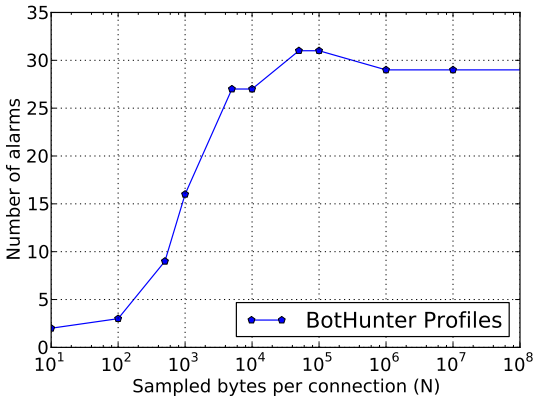



**Figure 8.9:** BotHunter Alarms

number of alarms for these eight hosts[1]. One machine generated a total of 12 profiles, all others have four or less profiles. It is unclear under which circumstances the correlation engine decides when to split a profile for one machine into multiple profiles.

The total number of alarms that must be inspected by an operator decreased dramatically compared to the number of total events that were produced by Snort. Instead of having to examine over 20,000 alarms, only 29 profiles must be inspected in order to determine false alarms. Unfortunately, we are not able to estimate how many false negatives BotHunter produces. However, this estimate is not in the scope of the work as we are interested in whether the sampling degrades the detection rate of the engine.

In order to determine the impact of the sampling on the results of the correlation engine, we prepared a number of alarm files. Each of them contains only the alarms that would have been sampled if a certain sampling limit were applied. A total number of ten sampling limits from 10

1 We replaced the actual IP addresses with place holders due to privacy reasons.

bytes to 100 MB are included in this evaluation. We start a new run of the correlation engine for each sampling limit.

Figure 8.9 shows the detection rates in the profiles depending on the sampling limit. The figure shows that only very few profiles are generated without the involvement of payload, e.g. by connections to known malicious IPs or domains. Another important observation is the increasing number of profiles with an increase of the sampling limit. As with the individual alarms, we can see that a 10kB sampling limit is sufficient to retrieve most of the profiles. With a 10 kB sampling limit, we receive a total of 27 alarms, hence only 2 profiles less than without sampling.

Interesting artifacts in our measurements can be found with a sampling limit of 50 kB and 100 kB. The application of both sampling limits results in more profiles than the application of larger sampling limits or no sampling at all. We suspect the inner decision making of BotHunter's correlation engine to be responsible for the effect. Profiles are generated while the alarms are consumed, i.e. not at the end of the alarm file. Hence, BotHunter decides on every incoming alert if a profile is created or not. This decision is most likely based on the sequence of incoming alarms, and we suspect the addition of certain alarms with an increased sampling limit results in the change to the numbers of profiles.

From our evaluation, we can conclude that the most relevant alarms for the BotHunter's correlation engine can be found at the beginning of the biflows of UDP and TCP traffic.

## 8.3 Conclusion

In this Chapter, we examined the applicability of our proposed sampling of the beginning of biflow communication for the analysis task of detection of worm and botnet traffic.

We used a well-known signature-based detection system to examine artificially generated traffic as well as live traffic. The generated traffic traces contained real botnet communication of malware that was executed within a controlled dynamic malware analysis platform. Incoming and outgoing communication of the executed bots was recorded and subsequently analyzed using vendor- and community-provided rule sets. We found that most of the TCP alarms in the malware traces can be found with very small sampling limits: 3 kB of payload were sufficient to retrieve most of the alarms.

Afterwards, we extended our analysis of the impact of the proposed sampling to live analysis of real-world traffic. We deployed our traffic analysis at a border gateway of a large university network and determined the impact of sampling on Snorts payload analysis. Throughout this analysis, we determined the sampling limits of alarms that were necessary in order to retrieve the alarms. We found that 10 kB of payload are sufficient to retrieve most of the alarms using an EmergingThreat rule set, as well as a rule set that was specially crafted for botnet-related traffic detection. For UDP,

however, we determined that large sampling limits are required to retrieve a comparable share of the overall UDP alarms.

We examined the resulted alarms with an automatic correlation engine, which tries to find multiple infection signs of botnet traffic. It is able to reduce the number of alarms that must be examined by an operator to a great extend. We determined the effect of our sampling on the results of the correlation engine and found that most of the alarms that are considered relevant by the correlation engine can be found at the beginning of TCP connections and UDP biflows.

# 9 Analyzing Caching Benefits for YouTube Traffic

**Notice of adoption from previous publication:** The text in this chapter is a modified version of the text previously published in

- Lothar Braun, Alexander Klein, Georg Carle, Helmut Reiser, and Jochen Eisl. "Analyzing Caching Benefits for YouTube Traffic in Edge Networks - A Measurement-Based Evaluation," in Proceedings of the 13th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2012), Maui, Hawaii, April 2012. [4]

The author of this thesis conducted the passive measurements and analyzed the properties of the data sets. He contributed significantly to the analysis of the video parameters and the simulations of the caching benefits.

## 9.1 Introduction

The past years have seen an ascent in the usage of web services and user generated content. HTTP has become one of the dominant protocols in current networks, both seen from a large Internet wide view [186] as well from a local networks' point of view [187, 105]. One flavor of user generated content that has come to a special interest of network operators and network researchers is video content. Video platforms such as YouTube, are known to be responsible for a large share of the overall amount of traffic exchanged in certain networks. A recent study from Labovitz et al. [186] quantifies this share to be as large as 20–40% of all HTTP traffic. With YouTube being one of the most popular video sharing applications, researchers conducted a lot of work that aims at understanding the platform and its impact on network traffic.

Video services such as YouTube are challenging for network operators, as these contribute a large amount of traffic in their (access) networks. Furthermore, video traffic is expected to experience a strong increase for mobile device usage [9]. Video content is typically static, and video popularity on platforms like YouTube is assumed to be Zipf distributed (except for the tail) [137, 188]. Content with these properties has the potential of large benefits from caching. However, several properties of YouTube traffic might have negative impact on cache hit rates and cache performance: One important factor is video popularity. Global popularity of YouTube videos, measured in view counts,

must not necessarily match local peculiarities in a specific operator network [135]. Popularity distributions may differ slightly depending on the network that a cache has to serve.

Factors such as video encodings or user behavior can impact cache performance, too. YouTube introduced several video encoding formats for its videos starting in 2007. Due to these different encodings, otherwise equal content is transformed into a complete different (as seen from a caches' perspective) video. Users can switch between encodings while watching videos, they can abort video download before completion, or can start watching a particular video from a certain position in the video file.

Our work aims at quantifying the caching potential of video traffic in end-networks in order to provide real-world data for network operators on caching benefits and hints to cache vendors on how to build their caches. We picked the YouTube video platform for our study because it is one of the major platforms for user generated video distribution, and has therefore received a lot of attention from researchers, operators and network equipment vendors. For our work, we monitored and analyzed all YouTube video traffic from an end-network with more than 120,000 users over a period of a month. We examine this traffic with respect to relevant parameters for network caches, and report our findings. Building on this traffic evaluation, we simulate an in-network cache for this network. We estimate the caching potential and the reduction of downstream network traffic from such a cache.

The remainder of this chapter is organized as follows. Section 9.2 introduces the parts of the YouTube application that are relevant for understanding our work. We explain interactions between clients and the YouTube video servers which have impact on the operation of a video cache. Section 9.3 presents our traffic monitoring setup and introduces the network in which our vantage point has been deployed. The obtained data sets span a period of one month and several general properties are discussed. Afterwards, we discuss several properties of this data sets which are relevant for caching, and present our evaluation on the benefits of an in-network video cache in Section 9.4. The chapter is concluded in Section 9.5 with a discussion of our results.

## 9.2 YouTube Explained

The YouTube video application has been studied in a variety of previous work. As the application is constantly developed and improved, several changes in the applications behavior could be observed during the past few years. Our discussion of the YouTube video application considers those parts of the application that have direct impact on caching and are relevant for understanding our work. For a more detailed and complete description of the YouTube video application, we refer the reader to the study by Finamore et al [67].

When talking about YouTube, one has to be aware that there is not a single web application for delivering videos, but two different ones. The first application targets PC systems, the other

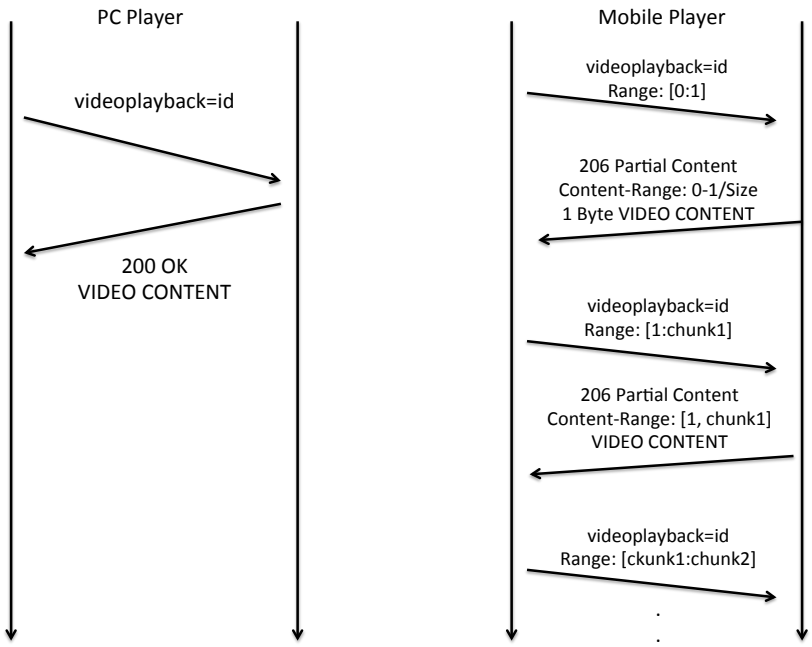



**Figure 9.1:** Video downloads from PC and mobile players

is optimized for mobile devices [67]. Both behave different on the network level during video download phase, but share common behavior before downloading the video.

Before a video can be requested from a video server, a user has to become aware of this video. Users can find videos from the YouTube portals sites (www.youtube.com for PCs or m.youtube.com for mobile devices) or via other websites which embed a video within an iframe. The portal or the embedded iframe contain a reference to a video in form of a unique video ID, which is then requested by the player.

Video download differs significantly between mobile- and PC-based devices. As first difference, PC players most often display videos using the Adobe Flash Plugin, which does not exist on several mobile devices such as the iPhone. It is also possible to watch videos directly using an HTML5 capable browser. However, this feature is currently only in a trial phase and users need to explicitly opt-in in order to download videos using HTML5. Mobile devices, which connect to YouTube from a WiFi network, usually do not have access to the Adobe Flash Plugin. Instead, they request and obtain a MP4 container (or similar) which includes the video directly.

The most important difference, however, is the process of downloading a video. Figure 9.1 shows downloading procedures for a PC and a mobile player. PC players, which are expected to have access to much memory and/or disk space, download a video using a single HTTP connection. Mobile players on the other hand download a single video using multiple connections where each connection is used to download a different chunk. HTTPs chunking mechanism is used to

implement this process. Chunks sizes are chosen by the client in the HTTP header, and servers reply with a *Partial Download* HTTP response. The first chunk size is always a request for a single byte of the video. Video servers respond with the first video byte along with the total size in bytes of the video. The client will then request subsequent video chunks of bigger sizes to download the video. Video encoding and start of the video are also requested by the client, which applies to both the PC and the mobile player.

## 9.3 YouTube Traffic Data Sets

This section builds the base for our YouTube traffic study. We introduce our monitoring setup that we used to observe and analyze YouTube traffic in in Section 9.3.1. Afterwards, we describe general properties of our obtained data sets in Section 9.3.2.

### 9.3.1 Monitoring Setup

Our vantage point was deployed in *Munich Scientific Research Network* at the same position as described in the previous chapters. The measurement setup was deployed on the same hardware as before, with the monitoring setup presented in Chapter 7. However, we did not apply our sampling algorithm to sample the beginning of the payload but analyzed all traffic from the YouTube infrastructure. We omitted the sampling in order to determine whether users watched only parts of the video, or whether they aborted the download in mid-download.

It was not possible to record PCAP traces for offline analysis due to the excessive bandwidth on the link. All traffic properties where calculated using online measurements for two reasons: The storage capacity of the system was limited so that it was not possible to store the many Terrabytes of video traffic that were observed during our monitoring period. Furthermore, the system is not capable to dump traffic to disk in line speed due to limitations of its disk performance.

We used tstat [65], which has built-in support for identifying YouTube traffic [67], as traffic analysis application on top of the setup. It was configured to observe and record all the information that was necessary for the analysis from live measurements. This approach significantly limited the amount of data that needed to be stored.

### 9.3.2 Data Set Properties

The monitoring setup was used to log information about the YouTube video downloads for a period of one month. We collected this data in order to be able to measure long-term statistics of caching relevant parameters. For our study, we focus on video traffic only. In particular, we do not list any statistics on the use of the YouTube video portal and related traffic (e.g. information on whether

**Table 9.1:** Monitoring Data Overview

| Property | Value |
|---|---|
| Start time | 16-Jul-2011 12:57:33 UTC |
| End time | 15-Aug-2011 13:47:10 UTC |
| PC Player | |
| # of video downloads | 3,727,753 |
| # of video IDs | 1,235,676 |
| # of videos with single encoding | 1,129,548 |
| # of videos with multiple encodings | 106,128 |
| Video traffic (PC player) | 40.3 TB |
| Mobile Player | |
| # of download connections | 2,480,703 |
| # of video IDs | 73,601 |
| # of videos with single encoding | 70,388 |
| # of videos with multiple encodings | 3,213 |
| Video traffic | 1.6 TB |

videos where embedded into other sites). We do not consider this information because we aim at caching video data, since it is supposed to be mostly static data and therefore qualifies for caching. Furthermore, video traffic accounts for the majority of YouTube related traffic and is therefore also the most interesting part of the YouTube related traffic for caching.

Table 9.1 describes the data set obtained throughout the monitoring process. The measurement was started in mid-July and was continually observing all video downloads until mid-August. We decided to distinguish between PC player and mobile player traffic, as shown in the table. PC player traffic was the dominant traffic type with respect to the overall amount of traffic. Furthermore, mobile players issue several connections for a single video download. Similar to [67], we use HTTP return codes for distinguishing between PC player and mobile players: Video requests from PC players are answered with a HTTP *200 OK* return code, while mobile video requests are answered by *206 Partial Content*.

Mobile downloads are only responsible for a small share of the overall video traffic (1.6 TB for mobile downloads vs. 40.3 TB for PC downloads) in our network. However, mobile downloads are responsible for quite a large number of connections. This is due to the download procedure which delivers a single video in multiple connections, as described in Section 9.2. We did not log the byte range requests from the clients requesting the actual chunks. Thus, we cannot give precise numbers about how many video downloads have been seen.

One interesting difference between mobile and non-mobile traffic can be found in the video encoding statistics, as shown in Figure 9.3. Most watched videos on a PC platform are transmitted as MPEG-4 AVC (H.264) encoded video with a 360p resolution which is embedded into a flash

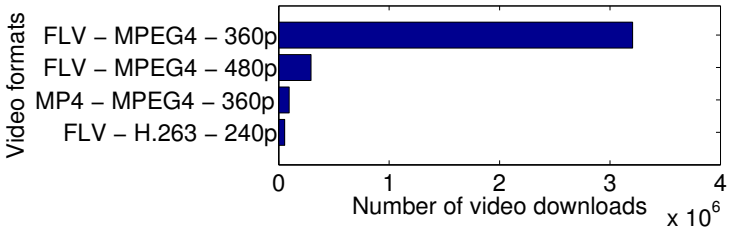



**Figure 9.2:** Video encoding for PC traffic

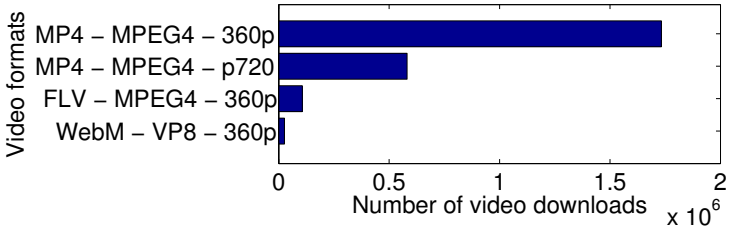



**Figure 9.3:** Video encoding for mobile traffic

container. Mobile videos are usually not embedded into flash containers, but are downloaded in a MP4 container. As for video content, the same encoding is used with a 360p resolution.

Hence, if the same video is watched with a mobile and a PC-based player, there is a high probability that a cache needs to deliver a completely different video (from a cache's point of view) for the same requested video. Operator networks that provide network access to an equal amount of mobile and PC-based devices, might therefore have to cache a lot of videos twice due to different encodings.

We were curious about the content types which have been requested most often by the users and therefore examined the most often viewed videos from PC players. The biggest share of the most popular videos where advertisements. Seven of TOP 10 videos can be placed in this category, with most of them being short ads for computer games. One particular popular video (Top 2) is part of a campaign advertising for a German pay TV station. We think that these videos were embedded into non-YouTube related sites and automatically downloaded by users who have visited those sites. The remaining two Top 10 videos are a news clip from CNN and a short fun movie. Advertisements or trailers for computer games, movies, cell phones, or beer dominate the Top 30 of the most often viewed videos. Each of these videos has a view count of more then 1500 views, and most of them are clips with a run time of less than two minutes. In the following section, we discuss several parameters that are relevant for caching this video traffic.

## 9.4 Evaluation

This section discusses video properties of the data obtained in the previous section. Relevant parameters for caching are discussed in Section 9.4.1. Section 9.4.2 evaluates the benefits of such

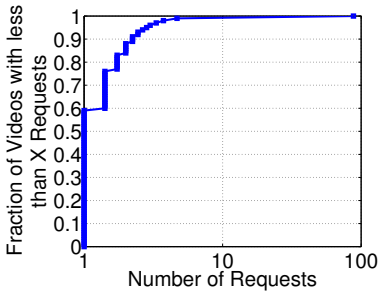



**Figure 9.4:** Requests per video

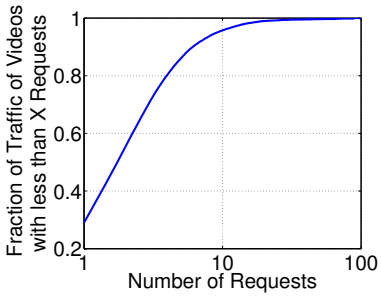



**Figure 9.5:** Overall traffic depending on the number of requests

an in-network cache. Due to the fact that PC player traffic is dominant in our observations, we will restrict our further discussion to PC player traffic. We will only discuss mobile device video traffic only if its properties differ significantly from the PC player traffic.

### 9.4.1 Relevant Videos Parameters for In-Network Caching

There are several important factors of video traffic that have large impact on a cache. These properties include video sizes, number of views of a single video or the inter-request times of multiple views, as caches can only provide benefit if videos or parts of videos are watched more than once. We distinguish between videos from a caches' point of view: Two videos are considered to be different if they have a different YouTube video id, or if they share the same video id but are encoded in different formats. In the following, we use the term video to address unique video content. Furthermore, the term request corresponds to a partial or full request of a video, while the term view indicates a full download of a video.

Figure 9.4 presents the share of videos out of the observed videos that have up to a particular number of requests. Our data reveals that about 60% of all videos are only requested once within our measurement interval.The remaining 40% of the videos can be cached and delivered from the cache to other clients for subsequent requests. The majority of the videos have been requested ten or less times, but some of the videos are watched several hundred or even thousand times. On average, each video is requested 2.7 times.

The huge share of videos that are viewed only once or a couple of times could lead to the assumption of only little potential for caching. However, if traffic volumes are considered, different trends can be observed: Figure 9.5 plots the amount of video content delivered from the YouTube servers for videos that have less than a certain amount of views. The majority of videos that have been requested only once are responsible for only approximately 30% of the video traffic. Thus, 70% of the traffic is generated by videos which are viewed at least two times. Videos that are watched more than once account for the biggest part of the traffic, which emphasizes the potential of in-network caches.

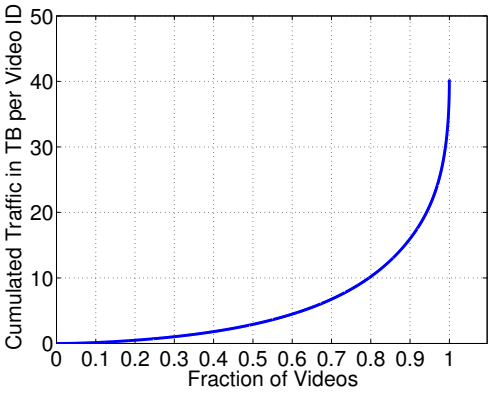



**Figure 9.6:** Total requested data of all videos

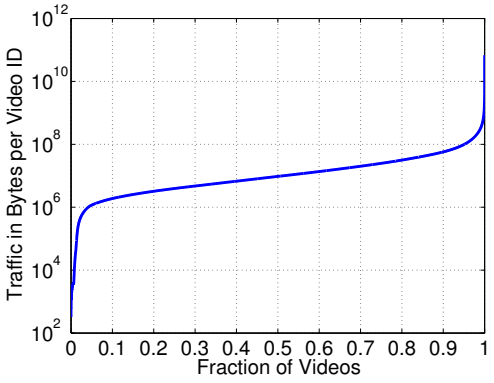



**Figure 9.7:** Total requested data per video

Figure 9.6 summarizes the influence of individual videos on the overall amount of download traffic. The graph shows the sum of the requested data per video sorted by traffic size in order to outline the traffic contribution of the individual videos. One can see that 80% of the videos are responsible for about 10 TB of traffic, while the remaining 20% account for 30 TB of all downloaded video data. By identifying and caching such high-profile videos, a cache can significantly decrease the amount of download traffic and achieve good cache hit rates without the need of a large storage capacity, since 20% of the videos generate 75% of the video traffic.

If we examine the share of traffic for individual videos, we can recognize a similar trend. Figure 9.7 plots the amount of traffic per video sorted by traffic size. The amount of traffic for each video is accumulated over all requests of the video during our measurement period. Only a very small fraction of videos contributes a very small share to the overall amount of video data. These videos were probably watched only for a single time and/or were aborted before being downloaded completely. 4.2% of the videos' download sizes are less than 1MB in data, while 4.9% of the video downloads generated more than 100 MB of traffic. Thus, more than 90% of the videos generate between 1 and 100 MB of traffic.

Request size and number of requests per video are the two factors that contribute to the amount of traffic that is generated by a single video. Very large videos are responsible for a large amount of traffic even if they are watched only a few times. Small videos that are viewed very often can also accumulate a lot of traffic over a large time interval. From a caches' point of view, videos with high request rates are most beneficial for hit rates and cache efficiency. Moreover, high request rates in combination with large request sizes would further reduce traffic from the YouTube servers which results in a decrease of link load between the edge network and the Internet. Table 9.2 lists the amount of traffic and the number of views by the TOP 5 videos which generated the most download traffic[1].

1 Video URLs: `http://www.youtube.com/watch?v=`$videoID$

**Table 9.2:** TOP 5 Videos

| TOP | Traffic (GB) | Request Count | videoID |
|---|---|---|---|
| 1 | 68.3 | 7758 | hJd9iCbpwuI |
| 2 | 33.3 | 4204 | zM41GVYYOMI |
| 3 | 16.4 | 22 | roFmDA2_yhg |
| 4 | 16.3 | 2826 | 60ZO8fVkfH4 |
| 5 | 14.5 | 4944 | Wsfgyyvs1tc |

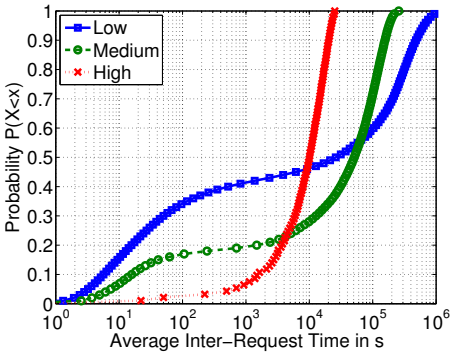



**Figure 9.8:** CDF of average inter-request time

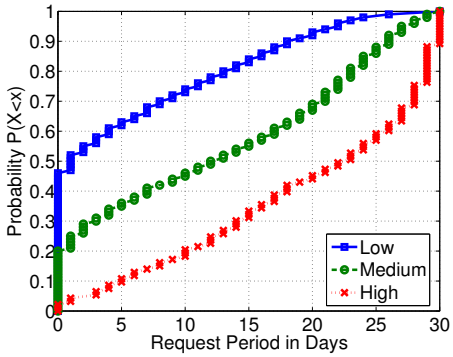



**Figure 9.9:** Request period depending on the number of requests

As one can see, the characteristics of the Top 5 videos in terms of generated traffic load and number of views differs significantly. The two videos with the highest amount of traffic generate almost the same amount of traffic than the next 8 ranked videos together. The heterogeneity of the videos is indicated by the number of requests, e.g. the video that is ranked third is only requested 22 times but contributes 16.4 GB of traffic whereas the video on fifth place is requested 4944 times. The same characteristics can be recognized for the top 20 videos in terms of requests. However, if we examine the overall video population, we can see that the number of requests is still a reasonable decision factor whether a video should be cached or not. This is due to the fact that average request size of all videos is 12.8MB.

Besides the amount of generated traffic per video, request patterns play an important role for the efficiency of a cache. Now, we take a closer look on the average inter-request time of the videos and the time period during which the videos were requested. Due to the heterogeneity of the videos, we decided to evaluate both characteristics for different groups of videos. The videos are grouped according to the number of requests which reflects their popularity. We defined three popularity groups: Low, Medium, and High. Videos are assigned to these groups based on whether the number of requests is in one of the following groups: $[2,10[$, $[10,100[$, $[100,\infty[$.

Figure 9.8 plots the average time between subsequent requests for these groups. It reveals that this average inter-request time of videos differs significantly depending on the video classification. 95% of the average inter-request-times of high popular videos are between 1000 and 10000 seconds.

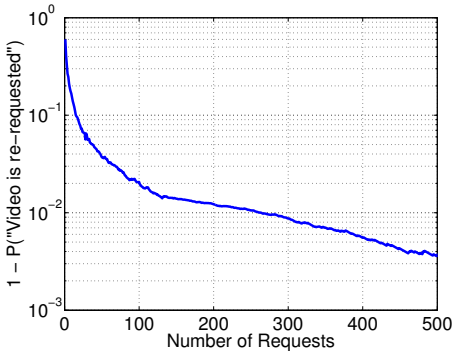



**Figure 9.10:** Probability for a video not being re-requested

This is in strong contrast to those with less requests: videos with low or medium popularity have a much higher variance of their inter-request times. The majority of these videos have an average time between requests of around 30 seconds.

We assume that this represents a typical value for non-popular videos that are posted in social networks. Thus, it is likely that friends will request the posted video resulting in a couple of full or partial downloads within a rather short period of time. In addition, a second peak can be recognized for videos with less than 100 requests for an average inter-request time of approximately one day.

Another important characteristic for caches is the time difference between the first and the last request of a video. We are able to calculate this time for our data set. It should be noted that the maximum request period is limited by the measurement period. The cumulative distribution function of the request period of videos with low, medium and high popularity are shown in Figure 9.9.

The figure points out that videos with a smaller number of views tend to have a shorter request period. 44% of videos with less than 10 requests have a request period of less than a day. This share decreases for videos with medium and high popularity to 20% and 3%, respectively. More than 50% of the high popular videos have a request period of more than two weeks. Almost 12% of all videos were requested over the whole measurement period which shows that a significant amount of videos are popular over a long time-period.

Cache sizes are very important for the estimation of caching benefits. Due to limitations in cache sizes, videos that are no longer watched need to be removed from a cache as soon as its disk is no longer able to store new videos. A video should not be removed if the probability for a subsequent request in the near future is still high. An indicator for estimating the probability of a future request is to examine the history of requests. For this reason, we evaluated the probability that a video is requested at least one more time depending on the number of previous requests. Figure 9.10 shows the complementary probability of this event in order to provide a higher readability.

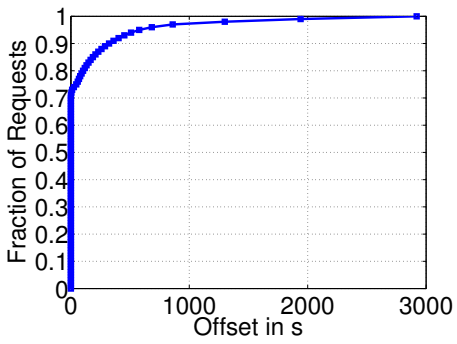



**Figure 9.11:** CDF of request offsets

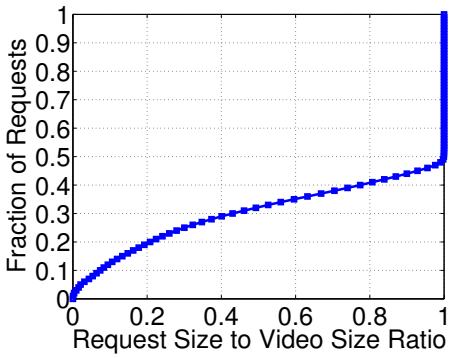



**Figure 9.12:** Request sizes

The probability that a video is requested at least one more time increases with the number of requests. The re-request probability of a video that was requested one time in the past is already 60%. This probability increases to 86% for videos that were requested 10 times and exceeds 98% for videos that were requested more than 100 times. The trend suggests that this probability converges against 99.9%. However, the number of videos with such a high number of requests was too low during our measurement period to support such a statement with a sufficient level of significance.

YouTube users do not necessarily watch videos from the beginning since embedded YouTube videos can directly jump into a specific part of video by specifying a starting offset. Furthermore, if a user forwards a video to a not yet downloaded offset, a new TCP connection will be opened which starts a new download beginning from the chosen offset.

Figure 9.11 shows the CDF of the offset of all requests. The figure reveals that 72% of all requests have an offset of zero. This means the majority of the users request the beginning of a video, which is very beneficial for caches. Only 3.5% of all requests have an offset greater than 1000s which results from the fact that the average video duration is 331s.

In addition, users can abort a video download before the video has been downloaded completely. This can happen for several reasons, such as the user experiences bad download quality or is not interested in the video [67]. Therefore, we evaluate the behavior of the users by calculating the fraction of request size and video size in order to track how much of the video the user has watched. The results are plotted in Figure 9.12. The figure shows that more than 50% of the requests download the complete video. Another 20% still download almost half of the video while only a very small fraction requests a small part of the video.

Videos that are not watched completely do not need to be fully cached. A cache has to decide whether the complete file is downloaded when a new video is requested, or if it only stores the requested bytes. We therefore examine whether certain parts of a video are watched with a higher probability, e.g. if the beginning of a video is more likely to be watched than the middle or the end of the video. Thus, we divided each video into chunks and calculated the probability for each chunk

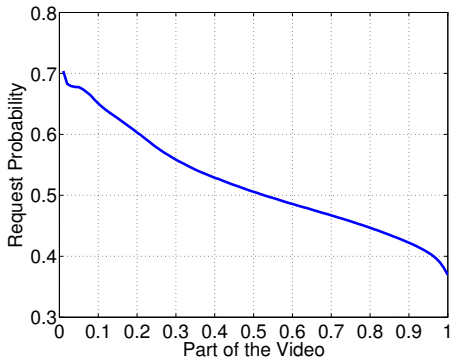



**Figure 9.13:** Request probability of different video chunks

to be downloaded. Offsets in a video request to the YouTube servers are defined in milliseconds. However, for caching purposes, we are interested in byte chunks. Hence, we need to calculate byte offsets. For this calculation, we use meta information from the flash containers that provides information such as total video length (bytes and duration). Our log files did not provide this information for other containers. Thus, we can only evaluate the chunk information for videos which where embedded in flash. Figure 9.13 shows the probability for different parts of the video to be watched in a given download.

We observe that not all chunks are viewed with the same probability. Video parts from the beginning of the video are more likely to be viewed than the latter parts of the video. The probability for a chunk to be viewed is decreasing with its distance from the start of the video, which is probably due to the fact that users abort a video before it is completely downloaded. We will study the effect of this finding in the following from a the view point of a cache. If a cache loads and stores chunks that are not delivered to the users at all, then it will create unnecessary traffic to the YouTube infrastructure. Furthermore, this content fills up disk space which can then not be used to cache relevant chunks.

In theory, one would expect very high probabilities for the beginning of a video to be watched most often. This expectation is based on the fact that video starts in the beginning is the default behavior when videos are played on the YouTube portal. One would further expect this probability to decrease as users realize that they are not interested in a video or find a more interesting video in the related videos.

However, our data suggests that this is not true. The reason is probably that a lot of videos in our data set are not accessed via the YouTube video portal but embedded into other sites such as social networks. This embedded video does not have to start with the beginning of the video. Instead, it can be linked with a parameter that specifies an offset into the interesting parts of the video. Another explanation for this effect can be the behavior of the player: Whenever the user forwards the video to a position that is not yet fetched from the server, the player will close the current

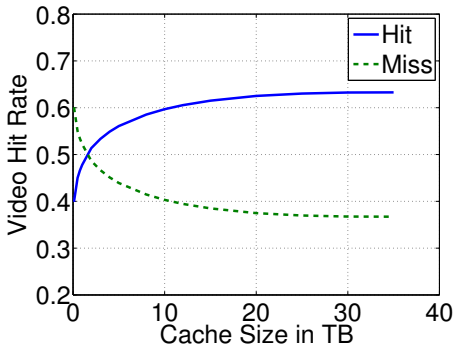



**Figure 9.14:** Request characteristic: Video Hits

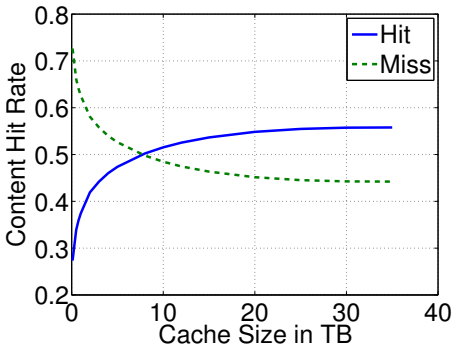



**Figure 9.15:** Request characteristic: Content Hits

connection. Afterwards, a new connection is opened with a request that contains an offset to the new position in the video.

These findings conclude that only the very last part of the video has a smaller probability of not being requested. When looking at the statistics of which parts of each video have not been requested at least one time, we could only find less then 2% of video parts, which have not been requested at least once. We therefore conclude, that a cache should always try to cache complete videos and not only certain parts of a video as there is a high probability that the other parts could be requested in a subsequent request.

### 9.4.2 Caching Benefits

For our evaluation of caching benefits, we use our measurement data as input for a simulation. Our simulation aims at answering the question: ”What if a YouTube video cache had been deployed in the network during our measurement period?” We calculate benefits that could have been provided by different caches and caching strategies.

Caching strategies that define how videos are downloaded and replaced are very important. Another important factor is the disk size, which is a major limitation factor for cache performance. A caching strategy must decide for each user request, whether it will download the complete video or only those parts that have been requested by a user.

Zink et al. [136] propose to download complete videos upon user request and deliver subsequent requests from this video cache. They also propose a last recently used replacement scheme from the cache: If disk space is exhausted and a new video needs to be stored, the video that has not been requested for the longest time is removed from the cache.

We implemented a simulation of this caching strategy and plotted the video and content hit rates for various disk sizes. A video hit is a user request for a video, which can be successfully answered from the cache. Video misses are user requests for videos that need to be fetched from the YouTube video servers. The same is applied to content hits and misses. Here we consider how many bytes of

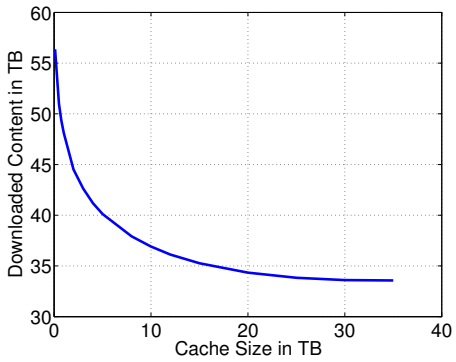



**Figure 9.16:** Downloaded content from YouTube video servers

the request needed to be fetched from the YouTube servers and how many bytes could be delivered from the cache.

Figure 9.14 shows the cache hit and miss rates for all video requests during our measurement interval depending on the cache size. We simulated caches with disks sizes between 100 GB and 35 TB, in order to determine hit an miss rates. Similar to Zink et al., we can see good hit rates. About 40% of all requests can be delivered from a cache with very small disk sizes (e.g. 100 GB). A cache with 2 TB disk space, could achieve a hit rate of more than 50%. Our maximum achievable video hit rate is more than 60% for a cache that is able to cache all requested content which corresponds to the video re-request probability for a video as shown in Figure 9.10.

However, a hit rate of more than 50% of the videos does not necessarily imply a high content hit rate. Figure 9.15 shows the content hit rate for caches of various sizes. We plot for each requested byte whether it could be delivered from the cache or whether it must be fetched from the YouTube infrastructure. Similar trends can be observed when looking at the hit and miss rates. However, 2 TB of disk space are not sufficient for a 50% hit rate in the cache. We need at least 8 TB in order to achieve a content hit rate of 50%. The maximum content hit rate is smaller than the video hit rate, but still exceeds 55%.

While these figures appear to be amazingly good, this caching strategy requires downloading the complete video. From our previous evaluation, we know that parts of the videos are not necessarily downloaded. Figure 9.16 shows the number of bytes that have been fetched from the YouTube video servers depending on the cache size. It can be seen, that this number is very high for small cache sizes and reduces to 33.6 TB with higher cache sizes. The reason for this is that all unique video content, if fully downloaded, results in 33.6 TB of traffic. However, users did not download this unique content completely, but only parts of it. This unnecessarily fetched data must be stored on disk and occupies disk space which is needed for videos that are requested completely or requested multiple times. For small cache sizes, many important videos are removed from the cache, and need therefore to be downloaded from YouTube several times for subsequent user requests. It is therefore important not only to look at cache hit rates, but also on the number of bytes which have

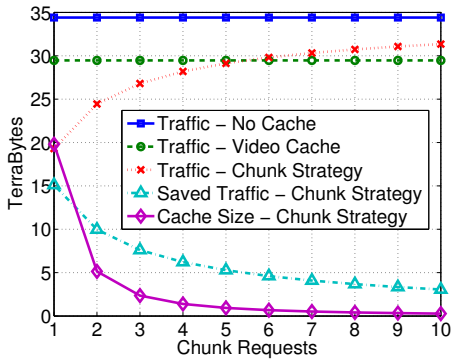



**Figure 9.17:** Chunked caching strategies

to be fetched from the YouTube video infrastructure. One more important conclusion, according to our monitored data, is that a caching strategy which fetches the complete content instead of the requested content, is not an efficient strategy.

Thus, we evaluated a cache which only stores content chunk-wise (chunk strategy): Videos are separated into 100 chunks, and chunks are only cached on user request. For the reasons outlined before, we can only consider flash content for this evaluation.

Therefore, the numbers for video data and requested content change: The complete size of the flash videos is 29.5 TB (compared to 33.6 TB for all videos). 9.7 TB of this video sizes where not viewed at all, e.g. due to premature download aborts. Storing these parts of the videos in the cache would unnecessarily occupy valuable disk space. User requests to YouTube for flash content sum up to 34.4 TB of video downloads, if no cache is used. A cache which downloads the complete video content if a video is requested (as simulated before), will download 29.5 TB of flash content from the YouTube provided that it is able to cache all requested videos. These two numbers are therefore the base-line for our chunked caching strategy.

A caching strategy has to provide mechanisms that decide when to store a chunk. Each chunk can be stored when it is requested for the first, the second, or more times. This decision has large impact on storage requirements and download traffic reduction of a cache. Popular chunks need to be downloaded twice, three times or more before any cache hit can appear, thus reducing the benefits in download traffic. On the other hand, waiting for a chunk to be requested several times before caching reduces the required cache size.

We evaluated the effects and benefits of a chunked caching strategy and plotted the results in Figure 9.17. The figure shows the cache sizes that are required and the traffic to the YouTube infrastructure, depending on the number of requests of a chunk before this chunk is stored. If we store chunks at their first request, a cache needs disk space of 19.5 TB for storing all chunks, and generates the same amount of traffic to the YouTube servers. Hence, when deploying such a cache, the amount of downloads from YouTube can be reduced by 15 TB. If we cache chunks on the second occurrence of a chunk, the required cache size drops to 5 TB (diamond markers), and the

amount of traffic to the YouTube servers increases to about 25 TB (cross markers). The amount of reduced download traffic drops by this 5 TB (triangle markers), since popular chunks need to be fetched twice. By comparing the results of the chunked caching strategy with the complete download strategy (triangle markers vs. dashed line), we can see that a properly configured chunked caching strategy performs much better than a properly configured strategy that downloads complete videos. Furthermore, the chunked strategy allows to deploy smaller caches to achieve this high caching benefits.

## 9.5 Discussion

In the study presented in this chapter, we used our measurement setup to analyze traffic between a large operator network and the YouTube video distribution site for over a month. During this time, we observed more than 3.7 million video downloads from PC devices and more than 2.4 million mobile video download connections. We found different properties, e.g. encoding and containers, for the different types of download connections. From a traffic perspective, our data set was dominated by PC devices which were responsible for 40.3 TB of download traffic compared to mobile devices which were only responsible for 1.6 TB.

Our analysis of the video proprieties therefore concentrated on the PC player downloads. This analysis found good local popularity values for YouTube videos. Good local popularity is an essential prerequisite for high caching potential.

Several videos where watched quite often in the same video encodings and resolutions within time frames that allow a cache to profit from such user download behavior. However, we also found user behavior that can have significant negative impact on caching performance and effectiveness. Behavior that can lead to negative impacts are users who prematurely abort a video.

Caches that do not take this behavior into account, experience caching performances penalties compared to properly configured caches. Hence, a cache must employ good caching and content replacement strategies. In order to determine the effects of different strategies, we run a couple of trace-driven simulations. They revealed that simple strategies can lead to good video hit rates with relatively small cache sizes. These simulations validate the findings by other researchers. However, we also examined content hit rates. Content hit rates were lower compared to video hit rates. A video hit rate of 50% could be reached with 2 TB of disk space. A comparable content hit rate required a cache size of 8 TB.

Previous studies did not consider bandwidth load on the access links. Our analysis revealed that a under-dimensioned cache that employs a content download strategy that loads the complete video can produce significant bandwidth on the Internet link. Such a content download strategy can lead more load on the access link, compared to a setup that omits the cache completely.

To circumvent these problems, we evaluated chunked caching strategies that restricts downloads to relevant chunks. We evaluated several different configurations of the chunked strategy and found that they provide very good performance for YouTube video content.

# 10 Analysis of the TLS/SSL X.509 PKI

**Notice of adoption from previous publication:** The text in this chapter is based on the paper

- Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. "The SSL Landscape - A Thorough Analysis of the X.509 PKI Using Active and Passive Measurements" in Proceedings of the 11th Annual Internet Measurement Conference (IMC '11), Berlin, Germany, November 2011. [3]

The author of this thesis performed the passive measurements. He developed both measurement setups, including adoptions to the Bro IDS software to allow for reliable extraction of certificate chains. He performed the analysis of the passive measurement data with regard to the TLS/SSL connections and their properties. Furthermore, he contributed to the analysis of the certificates and the interpretation of the results.

The author of this thesis rewrote the text of the original publication. As part of this rewrite, several sections that were part of the original publication have been removed because the author of this thesis considers them to be of minor importance. The removed portions of the text discuss the number of distinct intermediate certificates and chains, the statistics on certificate issuers, and the text on further certificate parameters. Additional information was added compared to the paper in order to clarify the argumentation. The section on service pervasiveness introduces new information that helps with interpreting the pervasiveness findings. Our discussion on Debian weak keys distinguishes between all and distinct certificates in order to be able to discuss the differences. We include the data set MON2 into our discussion of chain lengths. Finally, we included a second data set in our discussion of certificate quality in order to highlight the changes throughout our monitoring period.

Ralph Holz improved the methodology for certificate verification and validation after the paper was published. These improvements have impact on the certificate verification and validation. He performed a re-evaluation of the data sets. The re-evaluation only shows little variations, less than 1%, from the original results in the paper.

The numbers and figures in this chapter are based on the results of the re-evaluation. All figures in the chapter have been re-created by the author of this thesis. He changed the plot style of several figures in order to make them easier to read and understand.

## 10.1 Introduction

Protocols such as TLS and SSL are important in today's Internet, as more and more privacy sensitive information and confidential data is sent through the network. Organizations such as the IETF encourage the inclusion of security mechanisms into the software and protocol definitions. Specifically, the IETF requires all published protocol standards to contain a section on security considerations as part of the protocol specification [189]. The best current practices document in RFC 3552 [190] recommends the use of IPSec [191, 192] or TLS [193] to protocol developers to achieve basic security goals. Many IETF protocols that are designed to be used over the open Internet often rely on SSL [194] or some version of TLS [193, 195] in order to provide encryption, data integrity and entity authentication.

Both protocols rely on a Public Key Infrastructure (PKI) to provide authentication of end points. Besides authentication, the PKI is also used to distribute the keys that are required to set up encrypted connections between the communication end points. The PKI is built using the ITU standard X.509, which has been adopted by the IETF as an Internet standards track protocol [196, 197]. Protocol implementations and the underlying cryptography in TLS/SSL can be verified, providing some insight into the security they offer.

An assessment of the X.509 PKI is more difficult. The structure of the PKI involves multiple organizational entities and (potentially) human-driven processes that cannot be easily evaluated from the outside. These processes are used by Certification Authorities (CA) to create certificates. Certain steps must be done properly in order to ensure the security of the system, e.g. verification of entity identities. As the CAs' process implementations cannot be seen from the outside, end users are required to trust them without the possibility of independent review.

Problems in a single CA can have wide impact on the security of the complete PKI, e.g. in the PKI used for the WWW. A web browser is shipped with a number of trusted CAs that are allowed to sign certificates or allow other CAs to perform this action for them. Related work by the EFF analyzed the number of entities that are allowed to sign certificates for every host to be very large [145].

The EFF analysis found a total of 1,482 CAs that can create trustworthy certificates [145]. Each of these CAs is able to create certificates that are considered valid by a browser for any domain. A single compromised CA in this tree can therefore provide certificates that can be used for man-in-the-middle attacks for every known domain. While it is difficult to evaluate the processes within the CAs, researchers can try to analyze the certificates that are generated by them. Our goal is to obtain an understanding of the TLS/SSL landscape by the analysis of the deployment of the X.509 infrastructure. Traffic measurements are an important concept that can provide the necessary information to conduct a deployment study.

This chapter presents our evaluation of the Internet-wide deployment of the X.509 infrastructure for SSL and TLS. We collect TLS/SSL certificates by active and passive measurements and check their

security-related properties. Our data sets are enriched by publicly available certificate repositories (see Chapter 4.3 for a discussion of related work).

Combining our active data set with the data extracted from passively monitored TLS/SSL connections allows us to gain a thorough picture of the TLS/SSL landscape. Active measurements allow us to gain insight in the state of the *deployed* PKI infrastructure and certificates. Passive measurements can complement this view with information on the parts of the infrastructure that is *actively used*. Furthermore, passive measurements allow us to observe additional uses of TLS/SSL that are not visible using active scans, e.g. usage of TLS/SSL with protocols other than HTTP.

For this task, we perform packet-level measurements of TLS/SSL connections on a 10GE link of a large university network. The measurements observe around 250 million SSL and TLS sessions over a four-week period. They collected the exchanged certificates as well as security-related parameters of the communication, e.g. the negotiated ciphers.

The remainder of this chapter is organized as follows. Section 10.2 introduces X.509 and the structure of the PKI that is built upon the standard. The section highlights the relevant security parameters and properties, possible flaws in the processes that are responsible for setting these parameters, and the security implications of errors. Our data sets, their properties and the active and passive measurement methodology that we used to obtain the data are presented in Section 10.3. Section 10.4 presents the analysis and security related properties of the PKI information contained in the data sets. The chapter concludes with a discussion in Section 10.5.

## 10.2 X.509 Public Key Infrastructure

We introduce the fundamental properties of the X.509 infrastructure and the certificates that are used to authenticate TLS/SSL end points. Our focus is on a short summary of the properties that are relevant for understanding the following sections of the chapter.

The Public Key Infrastructure is built around Certification Authorities (CAs) which issue certificates. Each certificate contains information that can be used to identify the identity of an end point of a TLS connection. It is used to authenticate the TLS end point during connection setup. An end point has to provide the certificate as part of handshake, and the other side of the connection can use the information in the certificate to verify the identify of the other party.

Each CA needs to have a process for the generation of certificates. A crucial part of this process is the verification of the identity that is encoded into the certificate. In the following, the term identity will refer to the property that is encoded in the certificate properties, e.g. we refer to the domain name *www.google.com* instead of the company *Google Inc.* It is up to the CA to define the implementation of the identity check. The CA Browser Forum defines a document that specifies the *Baseline Requirements* [198]. Browser vendors like Mozilla define criteria that must be implemented by CA before they can be included into the root Store [199].

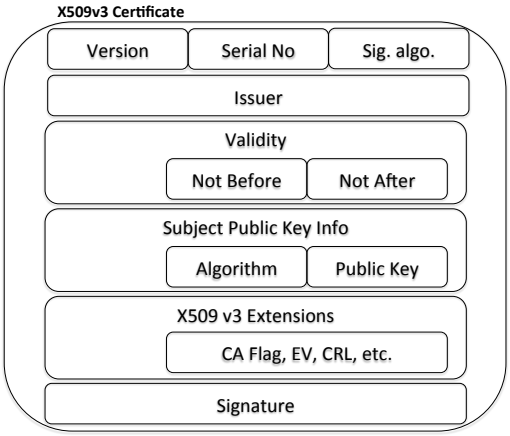



**Figure 10.1:** Schematic view of an X.509v3 certificate.

After the CA successfully checked the identity of the certificate requester, it will create the certificate. As part of the creation, the certificate is signed by the issuer to ensure its authenticity. Each certificate carries an additional important piece of information: the public key of the end point. The certificate thus provides a binding between the public key and the verified identity. This behavior can then lead to a transient binding of the certificate holder, who must possess the corresponding private key, and the identity.

Figure 10.1 shows a schematic certificate with the fields that are relevant for our analysis. A certificate contains information about the issuer – the CA that generated the certificate – as well as the identity that has been checked. This identity is encoded in the subject[1] of the certificate. It must be chosen in a way that allows TLS/SSL end points to valid the identity. Certificates are most often issued to servers and used to authenticate the server in protocols such as HTTPs, IMAPs, SMTPs, or POP3s. In most cases, the identity is encoded as the host name of the system, e.g. *www.google.com*. A identity can also match explicitly to more host names such as www.google.com *and* www2.google.com. Furthermore, it is also possible to issue a certificate for a wildcard – multiple not explicitly specified host names – to identify all hosts of a particular domain, e.g. **.google.com*. The TLS/SSL end points have to check whether the DNS name announced in the subject field matches the DNS name of an authorized end point or the DNS name of the requested host.

As mentioned before, the certificate contains the public key of the end point. It is used to set up the encrypted connection. Each certificate has a lifetime, which is encoded in the validity period. TLS/SSL end points are expected to check whether certificates are still considered valid when the TLS/SSL connection is established. In addition, X.509 allows the definition of certain extensions [200]. Most of the extensions are of minor importance for our analysis, and we will introduce some of them as needed throughout the text.

1 For HTTPs it is often encoded in the *Subject Alternative Name* or *Common Name* fields.

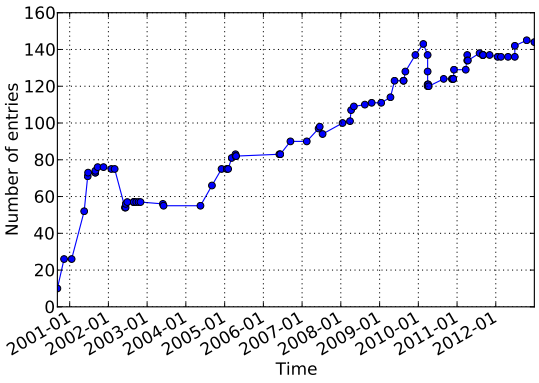



**Figure 10.2:** Growth of the NSS/Mozilla Root Store.

Certificates will be exchanged during the setup of the TLS/SSL connection. The certificates must be parsed in order to determine whether the communication is directed towards the desired system. If a server is presented with certificate by the client, it can determine whether the client is authorized to perform a connection. Some protocols, such as IPFIX [55], recommend such mutual authentication: A flow exporter is expected to present a certificate to a collector in order to enable the collector to check whether data is coming from an authorized exporter.

In many cases, however, only the server presents a certificate to the client. A client has to check whether the subject field of the presented certificate matches to the requested host name, e.g. during the establishment of an HTTPs connection: The browser needs to check if the subject field of the certificate matches the domain the browser wanted to connect to. Finally, the end points must check if the presented certificate was issued by a *trusted* CA.

X.509 does not require all certificates to be signed by a single CA, but allows for many CAs to be trustworthy. Thus, the certificate generation process may be distributed onto multiple entities. Furthermore, it allows to create private certification authorities, such as company-internal PKIs for private purposes.

End points are required to know which CAs are trustworthy, i.e. they need to know the certificate of the trusted CAs *before* the TLS/SSL connection is established. Trusted CAs must therefore be something that is externally configured on the system that participates in a TLS/SSL connection. Web browsers, for example, are shipped with a so-called *Root Store* that contains a list of trusted CAs. This list is provided as a list of certificates: The trusted CAs issue *self-sign* certificates that identifies them. These certificates are called *Root Certificates* and vendors include them into their *Root Stores*.

The process of including CAs into the root store is vendor-specific, and each vendor might trust different CAs. We analyzed the number of root certificates in the Mozilla Root Store. Figure 10.2 displays the number of active certificates in that store from 2001 to 2012. During this ten-year period, the number of trusted CAs rose from 20 to over 140.

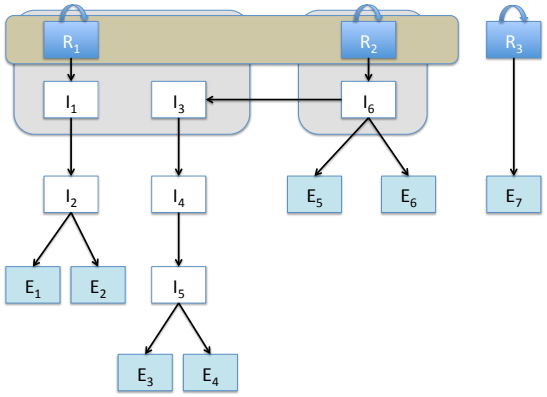



**Figure 10.3:** X.509 certificate chain examples.

The number of trusted authorities can be seen as one metric for the security of the overall PKI: *Any* trusted CA is allowed to create a certificate for *any* domain. An adversary who can compromise a Root CA would be able to issue a valid certificate for every domain, and would be able to use it for attacks against potential victims.

A compromise of a CA has happened before, for example as described in [201, 202]. An attacker hacked several CAs and was therefore able to create certificates for arbitrary domains. The first well-publicized incident involved only a small number of certificates, and resulted in blacklisting of the affected certificates by browser vendors [201]. In the other case of DigiNotar [202], more than 500 forged certificates have been created and could have been used in man-in-the-middle attacks. The root certificate of the CA was completely removed from the Mozilla root store after this security breach.

However, it is not necessary that an attacker compromises one of the certificates in the root store due to the chain of trust. The X.509 PKI infrastructure is not a flat infrastructure, but instead formed like a tree – since many CAs exist: a forest. Figure 10.3 shows a simple example forest.

As previously mentioned, root CAs reside at the top and issue certificates for themselves ($R_x$). Their certificates are usually not used for directly signing of end-user certificates. Instead, the root CAs issue intermediate certificates ($I_x$) that are then used to sign the end certificates. This is done for multiple reasons:

- Delegation of signing rights to other authorities
- Simplification of the removal of compromised keys
- Cross-signing between Certification Authorities

A CA may decide to delegate the signing of end-user certificates to other authorities. An example for such a delegation can be seen in the German research community: Many German universities

run their own computer networks, and connect to the "Deutsche Forschungsnetz" (*DFN*). *DFN* provides a network that inter-connects the universities and provides a connection to the rest of the Internet. Besides Internet connectivity, *DFN* offers additional services to the universities such as connection to the WWW X.509 PKI. Unfortunately, the certificate of the *DFN* CA is not part of most root stores.

However, *Deutsche Telekom* possesses a CA that is included in many browser root stores. *DFN* received an intermediate certificate that enables them sign certificates. The *DFN* then delegates the process of identity verification to German universities. Delegation of certificate creation rights therefore allows CAs, such as the Deutsche Telekom, to delegate the process of identity verification to authorities that are better equipped for the job.

Another important aspect is the exchange of keys in case of a compromise. Changing a certificate that is included in root stores of software is a complicated and time-consuming task. If a root certificate is compromised, then the replacement requires to change the vendor root store and updating all root stores at the end users, e.g. by deploying an updated version of the software. CAs are interested in not having their root certificates compromised. However, access to private keys is important for the signature process that is part of issuing end-host certificates. Intermediate certificates allow a CA to perform the task of signing end-user certificates with another certificate than the important root certificate. They can therefore keep their root certificate offline unless it is needed to sign an intermediate certificate, thus increasing the security of the root certificate.

Finally, intermediate certificates allow for easy cross-signing between authorities. Including a root certificate into a root store can be a difficult and long-lasting effort. Cross-signing allows CAs that are not present in one vendor's root store to be able to issue certificates that are accepted by the products of the vendor. In order to do so, it needs to get an intermediate certificate from another CA that is included in the vendor's root store.

Intermediate certificates lead to the concept of a *certificate trust-chain*. A TLS/SSL end point might not trust the intermediate CA that signed an end-certificate, but could trust the root certificate at the end of the chain. Consider the example from Figure 10.3: If a TLS/SSL end point needs to validate certificate $E_1$, it has to check the complete chain $I_2 \rightarrow I_1 \rightarrow R_1$ until it finds the root certificate $R_1$ that is included in its store. A chain may get arbitrarily long in theory.

While intermediate CAs provide the benefits mentioned above, they also introduce a problem: A vendor might try to limit the number of root certificates in his root store to trusted CAs. He might decide to include the certificate for CA $A$ because he thinks this CA is trustworthy. And he might decide to not include the certificate for CA $B$. However, if $A$ signs the certificate of $B$, then the vendor's products will accept certificates from $B$ as trustworthy.

If a CA certificate is signed by any root CA or any other intermediate CA, it will be trusted by the vendor that included the root CA. Increasing the number of CAs also provides a larger number of potential victims for someone who wants to compromise the PKI: If an adversary manages to

compromise any of the intermediate CAs, he will be allowed to create certificates for every domain and every host.

In order for a end-host certificate to be valid, several properties including the trust chain must be valid. Validity checks are performed by browsers and presented to end users in case problems have been found. However, studies showed that the most common reaction to browser warnings is to ignore the warning and proceed with the connection [203].

All these properties can lead to the conclusion that the X.509 infrastructure is a fragile construction. This claim or similar concerns has been voiced by different researchers [14, 204]. The goal of this work is to present an overview of the current state and the use of the deployment of the infrastructure in order to provide data on the real state of the quality of the deployment. In order to do this we perform extensive scanning of the TLS/SSL infrastructure and perform passive measurements of TLS/SSL connections. We compare the state of the deployment with the actual usage of the protocols and the deployed infrastructure.

## 10.3 Data Sets

Our analysis includes a number of different data sets that are obtained by different measurement techniques on different locations. The first type of data sets have been acquired by active measurements of TLS/SSL certificates. The second type of data consists of certificates that have been collected using packet-level measurements. Section 10.3.1 presents the active measurements. It describes our active measurements and the tools we employed to perform the certificate collection. Tools and capturing setups for the passive measurements are described in Section 10.3.2.

The resulting data sets and some of their properties are described in Section 10.3.3. An overview on the data sets is presented in Table 10.1. All sets that have been collected using active measurements have been published in order to make them available to the scientific community for further analysis [205]. The data collected by passive packet-level measurements could not be published due to privacy concerns.

### 10.3.1 Active Scans

In contrast to related work done by the EFF, we did not base our active measurements on complete IPv4 address space scans. Instead, we used the Alexa Top 1 Million Hosts list [148] to determine our scan targets. For each scan, we obtained a current version of the list at the date on that we start the scan. Alexa Top 1 Million contains a list of the most popular domains based on the ranking methodology of Alexa Internet, Inc.

Our goal was to obtain a list of popular domains. While the absolute accuracy of the list is disputable [206], we found it to be the best option to get a rough estimate of the popularity of a

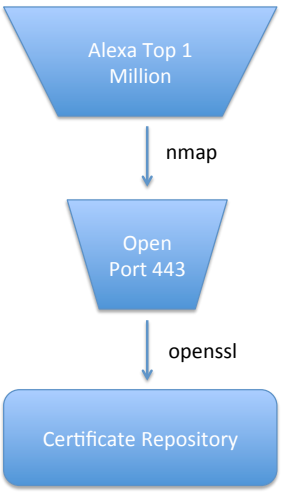



**Figure 10.4:** Active SSL certificate measurement process

site. We are especially not interested in whether a domain X and Y are ranked as the (n)th and (n+1)th popular sites, but more in whether they belong to one of the TOP 100 sites or have a popularity that is more to the bottom of the ranking. We slightly modify the list to match browser behavior: For each domain in the list, we choose the original domain name from the list, as well as a version of the domain that contains a *www* prefix if no such prefix is included in the domain name. Hence, we have a list of roughly two million (1,984,459) domains that are queried in our active measurements.

Figure 10.4 provides an overview of the active measurement process. Active measurements had two phases: In the first, the *nmap* [12, 207] network scanner was used to determine which of the listed domains operates an open TCP port 443, i.e. the port that is used for HTTPs connections. A total of three scans were performed. All domains with at least a single successful connection attempt were included on a list of potential TLS/SSL services. Hence, this first step resulted in a list of domains that offered service. It was afterwards used to collect TLS/SSL certificates. For this, we visited all the list members using the *openssl* tool that was configured to conduct a full TLS/SSL handshake. If the handshake was successful, we recorded and stored the full certificate chain, i.e. the chain of trust, along with the TLS/SSL connection properties that were negotiated during the handshake.

### 10.3.2 Passive Monitoring

We monitored all TLS/SSL traffic entering and leaving the *Munich Scientific Research Network* (MWN) in Munich, Germany. The monitoring was performed on the same hardware at the same observation point as described in the previous chapters. We obtained passive measurement data in two runs at different points in time. We improved our measurement software setup between the first and second run, whereas the hardware remained the same.

| *Short Name* | *Location* | *Time (run)* | *Type* | *Certificates (distinct)* |
|---|---|---|---|---|
| Tue-Nov2009 | Tübingen, DE | November 2009 | Active scan | 833,661 (206,588) |
| Tue-Dec2009 | Tübingen, DE | December 2009 | Active scan | 819,488 (205,700) |
| Tue-Jan2010 | Tübingen, DE | January 2010 | Active scan | 816,517 (204,216) |
| TUM-Apr2010 | Munich, DE | April 2010 | Active scan | 816,605 (208,490) |
| TUM-Sep2010 | Munich, DE | September 2010 | Active scan | 829,232 (210,697) |
| TUM-Nov2010 | Munich, DE | November 2010 | Active scan | 827,366 (212,569) |
| TUM-Apr2011 | Munich, DE | April 2011 | Active scan | 829,707 (213,795) |
| TUM-Apr2011-SNI | Munich, DE | April 2011 | Active scan | 826,098 (212,229) |
| Shanghai | Shanghai, CN | April 2011 | Active scan | 798,976 (211,135) |
| Bejing | Beijing, CN | April 2011 | Active scan | 797,046 (211,007) |
| Melbourne | Melbourne, AU | April 2011 | Active scan | 833,571 (212,680) |
| Izmir | Izmir, TR | April 2011 | Active scan | 825,555 (211,617) |
| São Paulo | São Paulo, BR | April 2011 | Active scan | 833,246 (212,698) |
| Moscow | Moscow, RU | April 2011 | Active scan | 830,765 (213,079) |
| Santa Barbara | Santa Barbara | April 2011 | Active scan | 834,173 (212,749) |
| Boston | Boston, USA | April 2011 | Active scan | 834,054 (212,805) |
| MON1 | Munich, DE | September 2010 | Passive | 183,208 (163,072) |
| MON2 | Munich, DE | April 2011 | Passive | 989,040 (102,329) |
| EFF | EFF servers | March–June 2010 | Active scan | 11,349,678 (5,529,056) |

**Table 10.1:** Data sets used in this work.

In order to deal with the large amount of traffic, both runs were configured to sample the first $n$ bytes of each TCP connection. This is sufficient, as the handshake messages including the X.509 certificates are exchanged at the beginning of a TLS/SSL session setup.

For our first measurement run, we captured the beginning of every observed bi-flow and dumped all sampled packets to disk, starting a new file as soon as a dump file reached 10 GB. Whenever a dump file was finished, the TLS/SSL connections were extracted offline. Due to disk I/O and disk space limitations, we were only able to sample the first 15 kB of each bi-flow.

The second measurement was conducted in April 2011 and employed online TLS/SSL analysis. For this run, we were able to use the multi-core aware setup presented in Chapter 7. With this setup, six applications of our monitoring software could be run in parallel. Each instance employed a sampling process that sampled the first 400 kB of each connection.

For TLS/SSL processing, we used the intrusion detection and protocol parsing system Bro [168] in both monitoring runs. Using Bro's dynamic protocol detection feature [208], we were able to identify TLS/SSL in a port-independent way. We used Bro-1.5 with some applied patches to the Bro code, which fix some issues and allow us to extract and store complete certificate chains from the monitored connections.

### 10.3.3 Data Properties

Table 10.1 summarizes the locations, dates and number of certificates in the different sets. Table 10.2 provides additional details for the passive measurements. Our data sets can be grouped into four classes:

Most of the active measurements were conducted from the research networks that our group had been located in: University of Tübingen and the Technische Universität München. Several active measurements were carried out in the time span from November 2009 to April 2011, i.e. over the time span of roughly 1.5 years.

One of the scans, in April 2011, was performed in a different way than the others: The openssl part of the scan was conducted using the Server Name Identification (SNI) extension of TLS [209]. A TLS connection that is established without this extension will request and receive the default certificate that is provided by the server. Using SNI, a client can specify the host name of the certificate it is interested in. This extension allows multi-homing of domains on a single IP address. The server can therefore provide different certificates depending on the domain the client is interested in.

Another part of our active measurements were performed April 2011, using PlanetLab [210] nodes from university networks in different countries. This was done in order to find difference in the TLS/SSL deployment in different locations of the clients. We wanted to check for location-specific handling of TLS/SSL clients. Content distribution networks (CDNs), for example, use DNS to route clients to specific data centers depending on the geographic location. Furthermore, some countries are known to employ censorship and our hope was to be able to determine whether some of the certificates are exchanged in order to allow for man-in-the-middle attacks during connection setups.

Th measurement process from the PlanetLab nodes differed slightly from the ones performed from Germany. On those locations, we omitted the nmap scans to determine the list of open service ports. Instead, we used the nmap result set from the scan from Technische Universität München, and only used the openssl wrapper to collect the certificates. This allowed us to shorten the scan period from the PlanetLab nodes.

Passive measurement made up for another part of our data set. The important distinction between certificates obtained from passive monitoring and those obtained by scans is that these certificates reflect that part of the PKI infrastructure that is not only deployed but also actually in use. Furthermore, it also contains certificates that are exchanged on ports other than 443.

Our data collection was conducted over two two-week periods in which we extracted all observed certificates from TLS/SSL traffic. In September 2010, we were able to observe over 108 million TLS/SSL connection attempts, resulting in over 180,000 certificates, of which about 160,000 were distinct. Our second run observed, during a similar time span of two weeks, more than 140 million

TLS/SSL connection attempts, which were responsible for about 990,000 certificates, of which about 100,000 were distinct.

The monitored network is a research network and hosts several high-performance computing clusters. We observed much TLS/SSL traffic from these computing clusters. Although most TLS/SSL connections are from HTTPs, IMAPs or POPs clients to their servers, many certificates originated from the computing clusters. The reason for this is the special usage of TLS in these environments: Certificates are used to authenticate user sessions, resulting in many short-lived certificates (mean validity around 11 hours). As Grid-related certificates are not comparable to those used on the WWW, we filtered them out in our analysis of certificate properties. The process of cleaning the data is described in Section 10.3.4.

The last class of data sets are the publicly available data sets that have been collected in related work. They contain data stemming from a different scanning approach: the EFF conducted a IPv4 address space scan that lasted several months until completed. This data set therefore contains a larger number of observed certificates, but lacks important information for certificate validity checks. Because the scan aims for IP addresses instead of domain names, there is no binding to the expected identity of the certificate owner. Our active measurement data sets contain this information.

### 10.3.4 Data Pre-Processing

Besides the standard PKI that is implemented by browser vendors for the WWW, there exist other PKIs. Some of them are private, e.g. for authentication inside companies. Others are public and used for special purpose applications.

Our passive measurements revealed a large number of certificates that are not accepted by the standard browsers, such as Firefox. Instead, they are used for authentication inside computing centers. The International Grid Trust Federation [211] operates an X.509 PKI that is separate from the one used for the WebPKI.

As our active measurement data set contained certificate data sets from HTTPs servers, we aimed at removing the Grid certificates from the passive measurement data in order to improve the comparability. Our passive measurement setup stored certificates and connection information. Unfortunately, we did not store a reference from the certificates to the TLS/SSL connections that exchanged the certificates. We were able to remove the Grid certificates based on the content of the certificates. Hence, our analysis of certificate properties could be done with a cleaned data set (see Section 10.4.2). However, due to the lack of reference from the certificate to the TLS/SSL connection, we were not able to identify the TLS/SSL connections that exchanged the certificates. Thus, it was not possible to remove the grid traffic from our analysis of TLS/SSL connections in Section 10.4.1. Although we cannot filter out Grid traffic in this case, we were still able to identify

some properties of Grid traffic by correlating the encountered IP addresses with those of known scientific centers.

The certificates could be cleaned from grid certificates using a content filter heuristic. In order to remove the unwanted certificates, we applied the following simple filter: if the certificate contains the word “Grid” or “grid” in the issuer field, i.e. the issuing CA, then the certificate is removed.

While we cannot guarantee that the filter is perfectly accurate, we cross-checked the results by verifying the removed certificate chains using the CAs in the Firefox Root Store. Our findings show that 99.5% of the trust chains in the removed certificates did not include any CA known to the Firefox Root Store. None of the removed certificates was considered valid, i.e. not a single removed certificate contains a valid chain that ended in a CA included in the Root Store.

## 10.4 Analysis of TLS/SSL Certificates

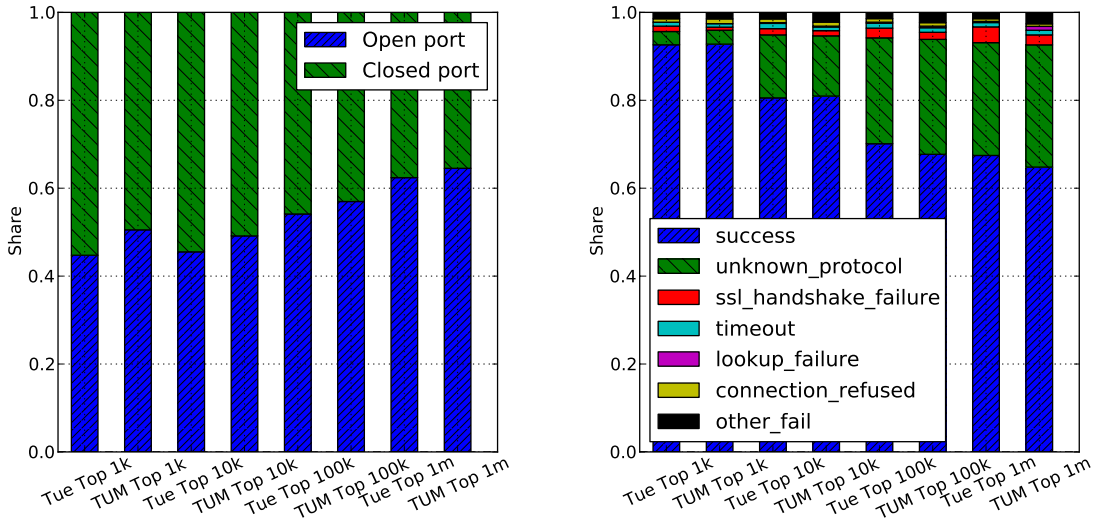
This section describes the results of our analysis of the collected data sets. Throughout the rest of this section we will make an important distinction: we either analyze the *full* data set of certificates, or we focus on the number of *distinct* certificates. The complete data set refers to the *deployment* of the certificates, i.e. if the same certificate is deployed on two different hosts, we account the certificate twice. In contrast, the *distinct* case provides insight into the quality of the *certificates* itself, i.e. the same certificate is only considered a single time. Each section contains a reference on the appropriate view point – *full* or *distinct* – that we are taking on the data.

### 10.4.1 Host Analyses

The first question that we wanted to ask is for the deployment of TLS/SSL in our active measurement data. This includes the questions on how many hosts do support TLS/SSL, and what connection parameters they negotiate during the handshake.

#### 10.4.1.1 TLS/SSL Service Pervasiveness

We analyze our scan results in order to determine how widespread TLS/SSL is deployed throughout the most popular sites. To determine how many hosts offer successful TLS/SSL connection establishment, we evaluate the results of the scanner described in Section 10.3.1. The results of this process are shown in Figure 10.5 for scans from November 2009 and April 2011. Our scanning is performed in two phases: The first phase employs a NMAP scan to determine which of the domains in our list provide service on the HTTPs port (compare Figure 10.4). A complete TLS/SSL handshake was only performed with the devices that had an open port.




**(a)** NMAP scan results

**(b)** Connection status

**Figure 10.5:** TLS/SSL connection errors for Tue-Nov2009 and TUM-Apr2011 scans

Figure 10.5a displays the results of the NMAP scans, Figure 10.5b shows the results of the TLS/SSL handshake for two of our scans. Both scans have been included into the figures because they were conducted at the beginning and the end of our monitoring period. Hence, they show how the deployment changed over our monitoring period. We also split our data sets into popularity categories – Top 1,000, Top 10,000, Top 100,000 and Top 1 million – in order to determine the usage of TLS/SSL throughout the different groups.

The first finding is that the use of TLS/SSL is more common among the Top 1 million than it is on the Top 1k: A larger share of the sites in the former category haven an open port than in the latter category. This finding is somewhat unexpected, as one would expect that high-profile sites are more likely to support TLS/SSL to secure the connections. A second finding is that share of open TLS/SSL ports was increasing between our measurements in 2009 and 2011.

A Closer observation of the services running on the open ports provides some correction to the initial unexpected findings of TLS/SSL support. While the share of open TLS/SSL ports through the overall Top 1 million list is larger than those of the more popular sites, real TLS/SSL support is not. Almost all of the Top 1,000 sites that offer a service on the HTTPs port provide TLS/SSL support that allows the successful establishment of a TLS/SSL connection. Only a small share of our connection attempts resulted in a problem. Figure 10.5b shows the different failures that we encountered throughout the handshake. The share of failures increases throughout the population of the Alexa Top 1 million list.

Roughly one third of all sites that offer service on the HTTPs port resulted in some kind of error. The most interesting finding is the high number of *Unknown Protocol* responses. They reflect an

| *Property* | *MON1* | *MON2* |
|---|---|---|
| Connection attempts | 108,890,868 | 140,615,428 |
| TLS/SSL server IPs | 196,813 | 351,562 |
| TLS/SSL client IPs | 950,142 | 1,397,930 |
| Different server ports | 28,662 | 30,866 |
| Server ports $\leq$ 1,024 | 91.26% | 95.43% |
| HTTPs (port 443) | 84.92% | 89.80% |
| IMAPs and POPs (ports 993 and 995) | 6.17% | 5.50% |

**Table 10.2:** TLS/SSL connections in recorded traces

error state in the TLS/SSL library that occurs if the received protocol messages do not comply with the TLS/SSL definition. In order to identify the reasons for the failures, we inspected a sample set of the affected hosts manually. All inspected samples offered the protocol on the HTTPs port: Plain HTTP without TLS/SSL, which points to the fact that there is a mis-configuration on the web server.

Other handshake failures do exist, however, their number if rather small compared to the *Unknown Protocol* failure Overall, about 800,000 hosts from the expanded Alexa list of 2 million targets allowed proper TLS/SSL connections on port 443.

The passive monitoring data shows many TLS/SSL-enabled hosts, as Table 10.2 reveals. An interesting finding is the high number of server ports found in the data set (more than 28,000 in MON1 and over 30,000 in MON2). This can be related to the observed Grid traffic, which we were not able to remove for this analysis (see Section 10.3.4).

As the table suggests, we see an increased usage of TLS/SSL in most metrics: The number of TLS/SSL connections, servers, clients and server ports increased within the half year that lay between our monitoring runs. Please note that both monitoring intervals did span a two-week period, and both were conducted throughout the semester breaks.

The increased usage of TLS/SSL could be related to the release of firesheep [212], which was released at Toorcon 2010 in October that year. Firesheep is a Firefox plugin that allows to sniff HTTP web traffic in wireless networks. The plugin received large media attention [213]. Afterwards, a number of high-profile sites such as Facebook or Twitter started deploying TLS/SSL as default and standard ways to work with the platform in early 2011 [213, 214].

Most of the TLS/SSL traffic was exchanged on well-known ports, with hot spots on the expected protocols: HTTPs, IMAPs, and POPs. The remaining traffic involved IP addresses that are assigned to scientific computing centers, and can therefore be considered to be related to grid computing.

#### 10.4.1.2 Negotiated Ciphers and Key Lengths

A good measure for protection of an encrypted connection can be found in the length of the involved keys and the choice of used algorithms [215]. Our passive measurements observed the

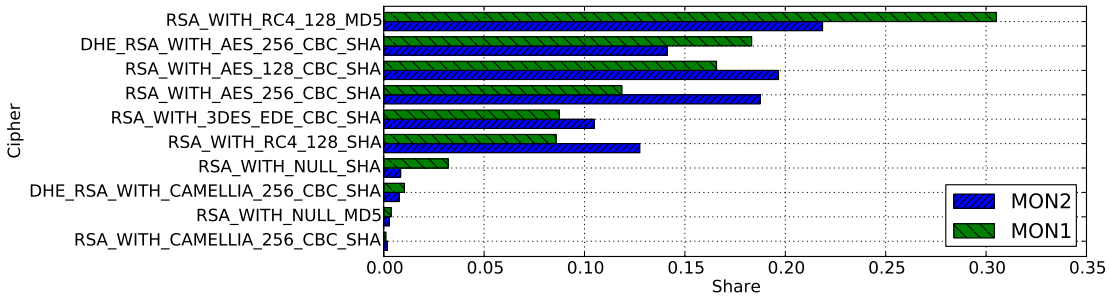



**Figure 10.6:** Top 10 chosen ciphers in passive monitoring data

non-encrypted handshake data, which includes a negotiation process of the ciphers. Hence, we were able to generate statistics on the used ciphers and key lengths that are used by the connections. We had to rely on the passive measurement data for this analysis: the choice of ciphers and algorithms is the result of a negotiation between the client and server. The decision on the used ciphers is done by server based on the offered ciphers by the client. Thus our active measurements do not show the results of in-the-wild negotiations, but only information on how clients would negotiate a connection if they are exactly configured as ours. All results of the active measurements are therefore biased and do not provide insight in real-world usage of TLS/SSL.

Figure 10.6 plots the results of TLS/SSL connection negotiations. A negotiation is a compromise between the client and the server on ciphers, key lengths and digests. The result of the negotiation therefore obviously depends the support choices on both ends of the TLS/SSL connection. We can see that the negotiation often resulted in a pick of strong ciphers in combination with good key lengths.

Older protocols such as 3DES are still used, however, in only a very small amount of connections. MD5 for Message Authentication Codes (MAC) is still in use, even though its use is discouraged: *"Due to significant progress in cryptanalysis, at the time of publication of this document, MD5 no longer can be considered a 'secure' hashing function."* [195]. The popularity of SHA-based digest algorithms seems to have been increased between our monitoring runs. The most popular combination of cipher and digest in both runs uses MD5, however we can see that its use decreased from over 30% of all connections to around 23%.

A very interesting class of TLS/SSL connections, although not very many, do not use encryption at all. They negotiated a *NULL* cipher, which means that they send the user data in plain text. MON1 had 3.5%, MON2 had 1% of all connections that picked this cipher. By manual inspection of some of the involved IP addresses that use this cipher, we can see that the connections are related to grid traffic. We suspect that TLS/SSL is used in this context to provide authentication, e.g. make sure that a user is allowed to connect to the computing center, but encryption is omitted for performance reasons.

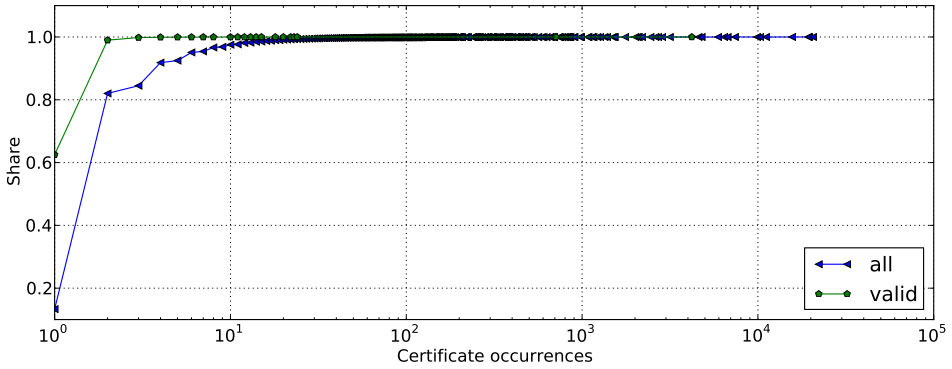



**Figure 10.7:** Certificate occurrences

### 10.4.2 Certificate Properties

Certificate properties are our view on the X.509 PKI for the WWW. This view includes trust chains that can lead to publicly known CAs, i.e. those who are in the root stores, or private CAs, which can only sign certificates that are not accepted by stock standard browsers. In this section, we investigate those certificate properties and try to understand the underlying X.509 PKI. For this analysis, we do not consider the grid-related certificates. We applied the techniques discussed in Section 10.3.4, to remove the appropriate certificates from our repository.

#### 10.4.2.1 Certificate Occurrences

Public key cryptography involves two keys: a public key and a private key. The public key can be known by everyone, while the private key must only be known by one party. If certificates are reused over multiple machines, then all of those machines must also know the private key in order to set up encrypted connections. If one of the machines that share certificates is compromised, then the private keys of all other machines are compromised as well. It would be therefore beneficial if every domain has its own certificates.

Multi-homing of web sites on a single machine can make it impossible to have separate certificates for the individual domains. By the time a TLS/SSL connection is established, the TLS/SSL layer must not necessarily know which domain is requested in the HTTP data. While there is the Server Name Indication extension of TLS/SSL which allows a client to request a certificate for a particular domain, the extension is not widely deployed. Furthermore, the creation of multiple certificates can cost more money than the creation of a single certificate. A larger number of certificates also increases the management overhead for administrators, as all of them need deployment. So it might be easier for them to share a certificate over multiple hosts.

In order to observe the magnitude of certificate sharing, we checked how many certificates were reused on multiple hosts. Figure 10.7 plots the CDF of certificate occurrences. The figure plots

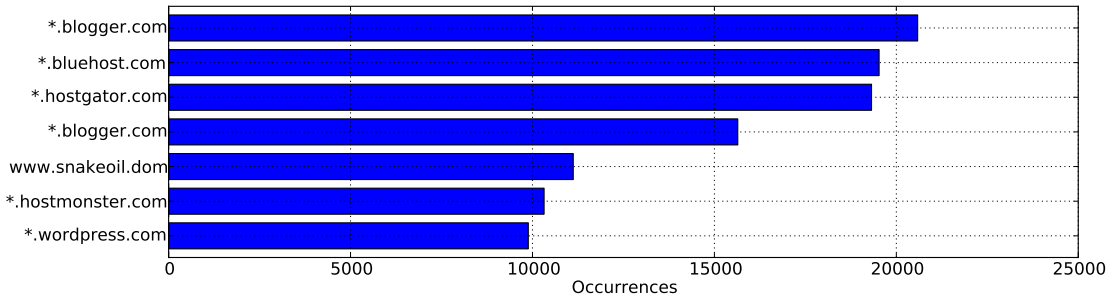



**Figure 10.8:** Domain names in most often re-used certificates

the share of certificates on the y axis that have been seen on $n$ or less domains for our active measurement set from September 2010.

We distinguish between two groups of certificates: the first group consists of all certificates from the pool. The second group only contains those certificates that are considered valid by a browser. We can see that valid certificates have a lower probability of being shared between multiple hosts: Over 60% of the certificates have only been seen a single time. Less then 20% of all certificates have only been seen on a single domain. This matches our expectations, as we would assume that if an administrator cares about valid certificates, he is more likely to ensure that the deployment of the certificates results in a secure setup.

Both curves show a very long tail, thus there exist certificates that are common for several thousand domains. We examined the content of these certificates in order to determine the domains that these certificates where issued for. Figure 10.8 shows the certificates that have been used by many domains.

One can see that most of the domains are issued as wildcard domains, e.g. **.blogger.com*. Three of the domains belong to blog services who create an individual domain for each blog, e.g. *blogname.wordpress.com*. Others belong to companies that offer hosting services. A special exception is the certificate for *www.snakeoil.dom*. This subject field is the default value from the standard certificate of the Apache Web server. It points to standard configurations that enabled HTTPs but did not go through the process of generating proper certificates.

Subject fields are crucial for validating certificates and therefore have a large influence on the validity of the certification chains. A certificate will not be valid if it encodes the wrong domain name in the subject fields. We inspect the validity of the trust chains in the following.

#### 10.4.2.2 Validity of Certification Chains

A number of factors influence whether a certificate is considered valid or not. The following list contains some of the requirements:

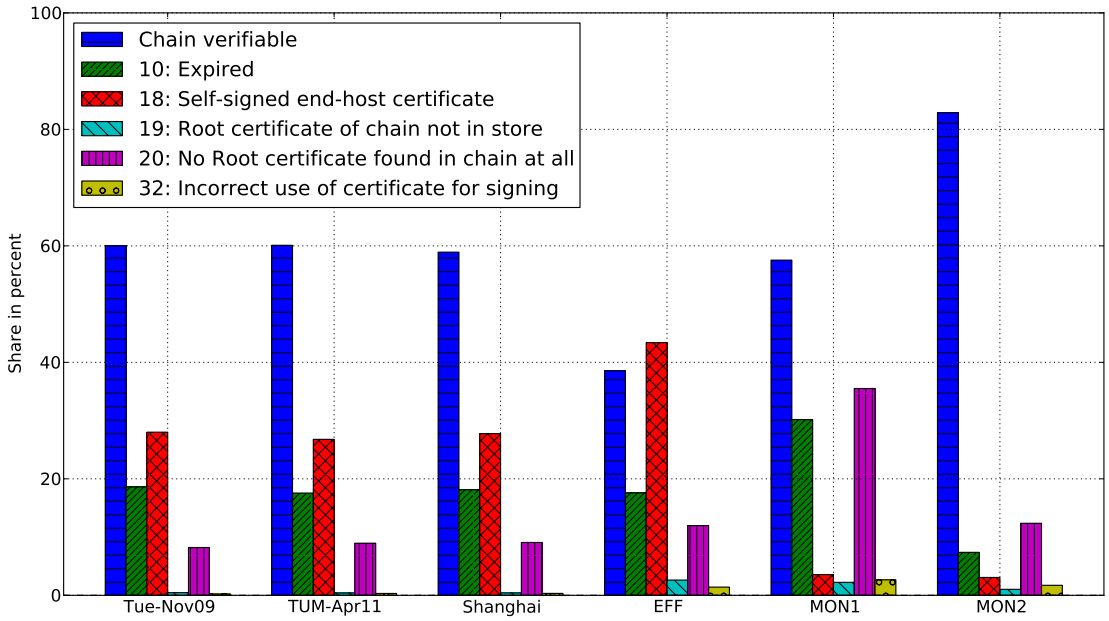



**Figure 10.9:** Error codes for chain verification

- The trust chain is complete.
- No certificate is expired or has broken signatures.
- The chain ends in a valid certificate in the Root Store.
- The certificate is issued for the correct domain.

Many factors can yield in non-valid chains, and thus in certificates that are not valid. We investigate certificate validity to determine how many of the observed certificates can be verified by a browser. As Root Stores are vendor-specific, the actual browser is important for determining whether a certificate can be trace to a certificate from the Root Store. We decided to use the Root Store of NSS library, which is included in some browsers such as Firefox, from the official developer repository from the time of the scan. This means that every scan data set was validated with the then-current Root Store. From these Root stores, we only consider those certificates that are used for the verification of WWW sites.

Verification is performed using the OpenSSL's *verify* command which performs a standard-conform validity check of a certificate. We use it to verify the full certification chain. This processing does not perform a crucial check: It does not match the presented domain name in the subject field with the desired domain name. We call certificates that can be traced to an entry in the Root store as *verifiable*.

Figure 10.9 shows the results for several of our data sets, including the errors that occurred during the verification process. A browser that performs the verification should present an appropriate

warning to the user. A user can then decide to ignore the error or do further investigation on the error reason. In the following, we explain the most important error codes that we received:

**Error 10 – Expired:** Corresponds to an expired certificate. For expiry verification, we compare the validity time in the certificate with the time at which we obtained the certificate. Hence, It does not reflect the time when we analyzed the data, but the time when we obtained the data. In case of the passive measurement data, we considered the end-time of the run as the expiry date. We did this because we didn't store the exact observation time stamps of the certificates due to disk space limitations.

**Error code 18 – Self-signed end-host certificate:** This error is generated for all certificates that do not have a trust chain at all. These are certificates that were used to sign themselves, i.e. the key that was signed is the same key that was used to perform the signing.

**Error code 19 – Root certificate of chain not in root store:** The certificate verification results in this error if it finds a correct chain, i.e. the trust chain is complete and can be verified to a proper CA certificate. However, the last certificate in the chain is not included in the Root Store.

**Error Code 20 – No root certificate found for chain at all:** Certificate verification results in this error if the issuer of the certificate could not be determined. This can result from the fact that the Root certificate is not sent in the chain and the Root certificate is not included in the Root Store. Hence, the error means that the chain that is sent by the server is incomplete.

**Error code 32 – Incorrect use of certificate for signing:** Certificate verification found a certificate in the chain that was used to sign a certificate but does not have the rights to sign a certificate.

The previous errors were the major errors encountered throughout the verification process. Very few certificates showed other problems: Some had very strange validity periods; others had broken signatures that could not be verified.

Figure 10.9 reveals verifiable trust chains for about 60% of all certificates in the active measurements. The most frequent errors are self-signed certificates with a share of about 25%, followed by expired certificates with 18%. Verifiable certificate ratios do not change significantly throughout the distinct certificates. The numbers were mostly stable throughout our monitoring period from November 2009 to April 2011. An interesting fact is that the domains on the Alexa list were not stable throughout this time period. More than 550,000 hosts in April 2011 from our extended list where not included in the November 2009 list. Our findings where not location-specific: measurements from the US and China did not differ significantly.

The figure, however, reveals differences between the active and passive measurements. We can see that the share of verifiable certificates differs between the two monitoring runs. The first monitoring run observed a fairly similar number of verifiable certificates than the active measurements. During the second run, however, the share of verifiable certificates was much higher. While the deployment

throughout the Alexa list did not change significantly, the monitoring data showed an increase in the ratio of verifiable certificates less than 60% to over 80%.

We compare our own measurement results with the results from the full IPv4 address space scan from the EFF. As presented in Figure 10.9, the differences are obvious: First, self-signed certificates are far more common in this data sets. This is an expected finding, as proper certification requires the spending of time and money. Administrators of low-profile sites may stick with self-issued certification or use the default values from the web server standard installation.

#### 10.4.2.3 Correct Host Name in Certificate

Certificate validity does not only depend on the inner structure of the certificate or the trust chain. A valid certificate is a verifiable certificate that is issued for the correct domain. Applications are responsible to perform a check on whether the certificate field encodes the entity that is requested by the user. Domains names can be encoded in the *Common Name* (CN) field as well as the *Subject Alternative Name* (SAN) field.

Our active measurement data sets have a mapping between the requested domains and the domains in the certificates. This information is not available for the passive measurement data: A host field within an HTTP request is only sent after the encryption is set up[2] We are therefore not able to determine the host from which the client would like to obtain its certificate from. We also cannot conduct this analysis on the EFF data, as the required ground truth is not available in the data set, either.

For the active measurement sets, we check whether the requested DNS name matches the appropriate fields, including wild card matches on subdomains. An exception to this rules is the wild card *, which would match every domain. We consider this to be an illegal name in a certificate. This behavior is coherent with the behavior of the Firefox browser, which also rejects certificates for this wild card.

In TUM-Apr2011, we found that the subject field for roughly 120,000 of the roughly 830,000 certificates matched the requested name in the CN field. When we considered the SAN content, the number of matching certificates increased to a little more than 174,000 Of these certificates, only around 100k (CN) or 150k (CN + SAN) had a complete valid chain of trust. This results in only around 18% of the collected certificates to be considered completely valid. The numbers didn't change significantly when we enabled the SNI features that allows us to specifically ask a TLS/SSL server of a certain certificate (the difference was 0.02%) The scans Tue-Nov2009, TUM-Apr2010 and TUM-Sep2010 result in a similar picture of 15%, 16% and 17% of valid certificates. We can see that the share of valid certificates increases over time, but only to a small extent.

2 It is possible to get this information if the SNI extension is used. But we did not log this information as part of our measurement if it was provided.

An important finding is that all data sets found that less 20% of the certificates deployed throughout the Alexa Top 1 Million list are valid. Hence, more than 80% of sites result in warning messages in browsers. We suspect that most of the failures are due to HTTPs being enabled on the servers, but not intended to be accessed publicly. Many users may therefore not encounter these sites or problems.

#### 10.4.2.4 Host Names in Self-Signed Certificates

Self-signed certificates are one way to simply deploy TLS/SSL for small or internal sites. They are in use by sites which do not want to invest time and money to go through the certification process at a CA included in typical Root Stores. If users want to obtain certificates without the involvement of a commercial CA, they can generate their own.

Generating X.509 CAs and certificates is a complicated task that can be difficult to get right. A high number of problems could therefore be expected for these certificates. One of the things that can result in non-valid certificates is failure to encode the proper host names into the certificates. We therefore checked the correctness of the CN field in the certificates in our SNI-enabled scan (TUM-Apr2011-SNI). Our findings are that about 99% of all analyzed certificates do not match the expected host name. The Subject Alternative Names was correct in about 0.5% of cases, but the feature was rarely used at all (little more than 1% of certificates).

#### 10.4.2.5 Extended Validation (EV)

X.509 was developed in order to provide an infrastructure that allows entity authentication. As discussed before, deployment of proper CA processes is important for the X.509 PKI to work as intended. This issue became an imminent risk due the increasing importance of the Internet for commercial activities. Business transaction on the Web became standard, and proper authentication therefore more important.

A baseline requirements guide exists, that defines basic requirements for domain validation [198]. They are limited to checks that for a specific domain, but do not require a verification of the identity of the organizational status of the domain holder. The CA/Browser-Forum published requirements that includes processes checks of the legal status of an organization [216]. CAs that comply with these guidelines are allowed to issue “Extended Validation certificates” (EV certificates).

Technically, these certificates do not differ from regular certificates. Thus, they do not provide any additional cryptographic security. The certificates instead include a *object identifier* that signals the EV property to browsers. Browsers are then expected to display this extended validation status to the user.

| *EV Status* | *Tue-Nov2009* | *TUM-Sep2010* | *TUM-Apr2011* | *Shanghai* | *Santa Barbara* | *Moscow* | *Izmir* |
|---|---|---|---|---|---|---|---|
| Yes | 1.40% | 2.10% | 2.50% | 2.56% | 2.49% | 2.51% | 2.50% |
| No | 98.60% | 97.90% | 97.50% | 97.44% | 97.51% | 97.49% | 97.50% |

**Table 10.3:** Deployment of EV certificates

Table 10.3 presents how many certificates contain the EV status over time. Our scanning period shows an increase in the usage of these certificates. We inspected the use of EV certificates within the top range of the Alexa hosts in TUM-Apr2011. Our findings are kind of surprising:

- TOP 50 – 5.17%
- TOP 100 – 8.33%
- TOP 1,000 – 8.11%
- TOP 10,000 - 8.93%

This contradicts the expectation that the EV status would be in use with more popular sites more often than with not-so-popular ones. The intuition is based on the fact that EV certificates are most often more expensive than standard certificates.

We found two reasons for this observation: Most of the Top 50 hosts belong to a single company that does not use EV certificates: Google. In addition, login servers for popular sites (e. g., Amazon or eBay) use EV certificates for the login procedure. However, these companies use different sets of servers for the log in than for the other parts of the business. Those other servers do not have EV certificates, but are used more often than the login servers. Hence, the login servers do not belong to the top 50 hosts.

#### 10.4.2.6 Length of Certificate Chain

Trust chains are an important concept within the X.509 PKI. It is beneficial for CAs to use intermediates for the reasons outlined in Section 10.2. However, they bear the potential of introducing more sources of error in the process. An increased number of entities that are allowed to generate certificates also broaden the attack space for potential adversaries. Long certificate chains also can potentially reduce performance due to the requirement to send and evaluate more certificates until a Root CA is reached. We calculate the length of the chains for non-self signed certificates for end hosts, by counting the number of not self-signed certificates minus the end-host certificate. This calculation overestimates chain lengths if certificates are sent with multiple chains. Our chain length represents the number of Intermediate Certificates, and can therefore be zero if no Intermediate is included in the chain for all certificates.

Figure 10.10 compares two of our scans from Germany, with data from one monitoring run and the Internet-wide EFF data. The large majority of certificates have a chain-length smaller than

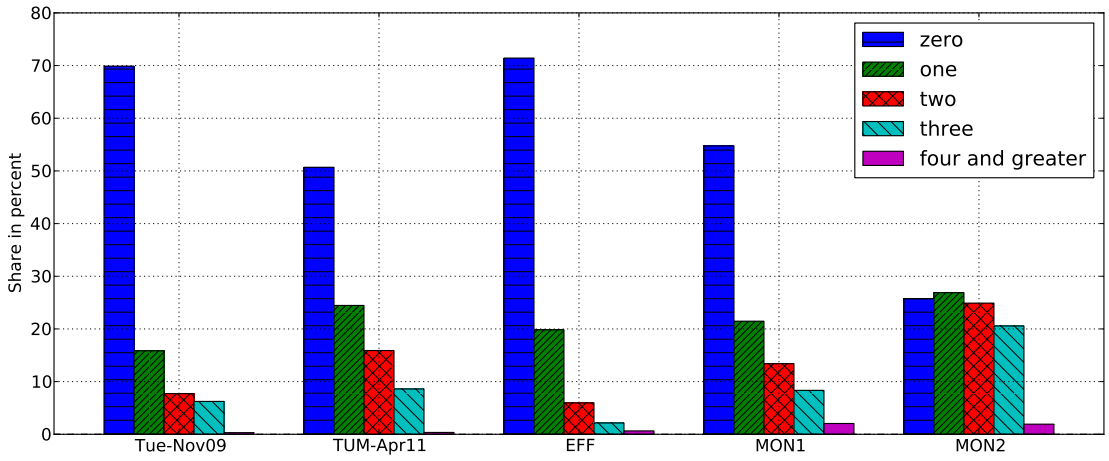



**Figure 10.10:** Chain lengths

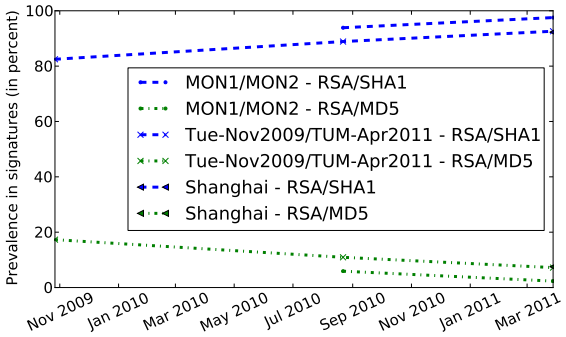



**Figure 10.11:** Popular signature algorithms in certificates.

three, with many certificates employing chain length of zero. This is due to the large number of self-signed certificates, or certificates that are created by a private CA. This number has largely decreased between November 2009 and April 2011, while the share of longer certificate chains increased.

MON2 differs from the other data sets. While the vast majority of all certificates have a chain length of zero, there are many more chains that have a length of one, two or three. The number of longer chains, i.e. four and more, is not significantly higher in MON2, though. An explanation of these findings can be found at a closer look at the certificates: We found a larger number of certificates in MON2 that belong to servers from companies such as Google, Microsoft, or Apple with a chain length or two and three. Those certificates have not been observed in MON1.

#### 10.4.2.7 Signature Algorithms

The past years have seen several new attacks on some hash functions. A prominent example is the MD5 function [217], but there also exist recommendations to substitute other stronger algorithms

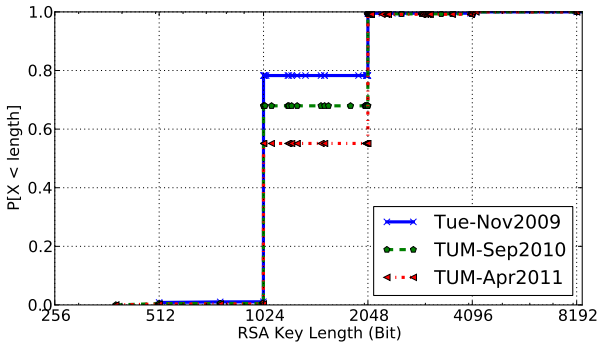



**Figure 10.12:** Cumulative distribution of RSA key lengths.

like SHA1 with stronger successors. [218]. Shifts away from MD5 have been expected after new research shows how to create valid faked certificates using hash collisions [219].

We analyzed our data sets with respect to the used signature algorithms in order to check if we can see changes in the certificate population that reflects this expectations. Figure 10.11 displays the use of algorithms for active and passive measurement data. It plots the deployment and use of the most popular combinations RSA/SHA1 and RSA/MD5 against each other. All other combinations have only seen in a minority of certificates (as consistent with the trends shown in Figure 10.6).

We can see that the combination of RSA/MD5 is declining over time, whereas the use of RSA/SHA1 is increasing. In 2009, about 17.3% of all certificates have been signed with a combination of an RSA and MD5 algorithms. Two years later, the figure dropped to 10%. SHA1 gained a similar share of certificates over the certificates.

The passive monitoring data shows similar trends. The overall use of the discouraged MD5 algorithms is even smaller than seen in the active measurement data.

Our conclusion here is that while MD5 is still sometimes used, it is indeed being phased out.

#### 10.4.2.8 Public Key Properties

Certificates do not only authenticate the end point, but they also provide public keys for the establishment of an encrypted communication channel. It is quite obvious that the used ciphers should be strong and they keys should have a good length. In 2009, for example, the authors in [220] show that RSA with 768 bit can be factored. NIST recommends moving from RSA-1024 to longer ciphers [221].

Key lengths found in our data sets Tue-Nov2009 and TUM-Apr2011 show an increase in the key length. Usage of keys longer than 1,024 bit increased by 20% while share of shorter keys fell by about the same amount. This trend is shown in Figure 10.12: The newer the data set, the smaller

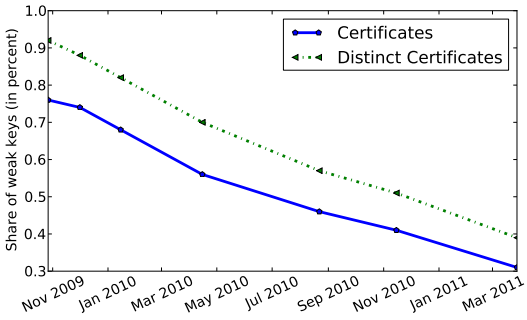



**Figure 10.13:** Debian weak keys in Alexa Top 1 million scans

the percentage of keys lengths for short keys. In general, this points to the fact that the number of longer keys increased.

The point indicators in the plot lines are placed on jumps of the CDF curves. They reveal unusual key sizes, i.e., key lengths that are not a power of two nor the sum of two powers of two. It can be seen that their share is negligible as the CDF does not change significantly at these places.

Besides key length, another property is important: Keys in different certificates should never be duplicated, unless the owner of the key is the same. A public key consists of two things: An exponent and a modulus. The most frequent RSA exponent we found in Tue-Nov2009 was 65,537. It was used in over 99% of all cases. The second most often used exponent with a share of 0.77% was 17. This finding is consistent with the passive monitoring data and scan from November 2011 (results differed by less than 0.5%).

Factorization can be a goal of attacks against the public/private key systems. This should be hard as given by the underlying mathematics, but implementation problems can turn this process into an simple exercise. A bug in the OpenSSL implementation of the Debian system had caused weak randomness, resulting in keys that could be easily pre-computed [222]. We investigated the occurrence of certificates that had been generated when the bug was included in the system. Figure 10.13 shows the development of the deployment throughout our monitoring interval. First, it can be seen that less than 1% of all certificate had been generated under the influence of this bug, both for all and the distinct certificates. Second, the share of such certificates is steadily declining. Notably, twenty certificates of those were valid.

Another issue are duplicate keys. No exponent/modulus combination should occur in different certificates. In TUM-Apr2011 we found 1,504 distinct certificates where this problem could be observed in different certificates. This number is similar to the one at the beginning of our scans in Tue-Nov2009, which was 1,544. While the OpenSSL bug could be a cause for this, we only found 40 (Tue-Nov2009) (10 in TUM-Apr2011) of these certificates where generated using one of the pre-computable Debian-bug related certificates.

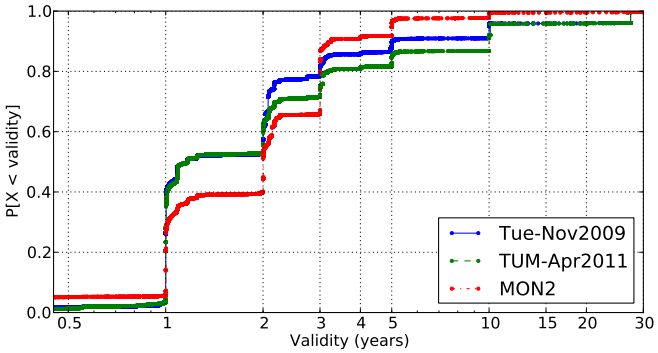



**Figure 10.14:** Comparison of certificate validity periods.

#### 10.4.2.9 Validity Period

Another important factor for certificate security can be found in the validity period. As cryptography is based on an arms race that provides security as longer as attackers are not able to factorize the keys, key lengths that where considered sufficient to provide security in the past are no longer considered a good choice. Validity periods should therefore only be set to a certain time interval in order to phase out old public keys.

Most of the certificates in our scan had a validity period of 12 to 15 months. This corresponds to a validity of a year, plus a grace period that can be used by operators to exchange the old certificates with new ones. Other popular lifespans are two, three, five, and ten years, as plotted in the CDF in Figure 10.14. Over time, the share of certificates with life spans of more than two years increased. This applies especially for certificates with a life span of ten years. Users of the infrastructure most often observe certificates with a validity of one, two, or three years. Certificates with validity periods longer than five years are rarely seen by users.

In the figure, we did not show all possible validity periods: Our data included certificates with very short, e. g., two hours, and very long periods up to 8,000 years.

#### 10.4.2.10 Different Certificates between Locations

One reason behind conducting active measurements from multiple locations world-wide was to analyze whether there certificate deployment changes between locations. We expected that this could happen for Content Distribution Networks that guide users to different data centers depending on their geographic location. Deployment of certificates can also be a time-consuming if servers in different data centers need to be updated. So, the deployment could be different depending on a number of factors.

| *Scan* | *Suspicious certificates* | *Differences to TUM-Apr2011* |
|---|---|---|
| Santa Barbara | 1,628 | 5,477 |
| São Paulo | 1,643 | 6,851 |
| Melbourne | 1,824 | 7,087 |
| Izmir | 2,069 | 7,083 |
| Boston | 2,405 | 5,867 |
| TUM-Apr2011 | 3,245 | — |
| Shanghai | 10,194 | 9,670 |
| Bejing | 10,305 | 9,901 |
| Moscow | 10,986 | 11,800 |

**Table 10.4:** Different and suspicious certificates

Another reason could be of malicious nature: a middlebox or router can intercept traffic and swap certificates transparently in order to stage a man-in-the-middle-attack. If an attacker is able to do this, he can read all encrypted traffic between the end points.

Our first investigation therefore aimed at finding certificates for certain domains that differ between locations. We labeled them as 'suspicious' if they have been observed as identical for most locations, but differ in one to three of other locations. Table 10.4 shows the results from each vantage point.

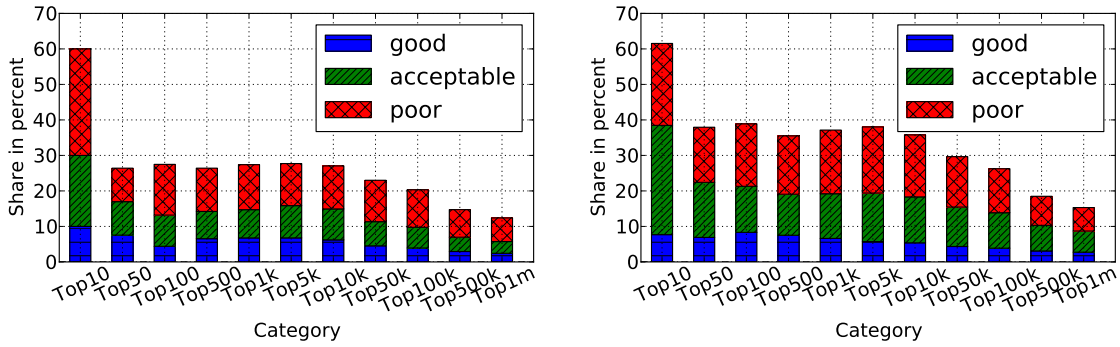
The table reveals a bias of suspicious cases toward our scans from China and Russia. However, this might be an artifact from localized CDN traffic. In order to dig into this, we closely examined data sets from Shanghai and TUM-Apr2011.

The two locations differ in 9,670 certificates, which is about 1% of the TUM-Apr2011 data set. From these, only 213 were in the top 10,000 ranks of the Alex list. The highest site had a rank of 116. Operators of higher-ranked CDN sites seem to properly deploy their certificates throughout the regions.

The differing Certificates from Shanghai where mainly not valid – only 521 certificate had correct chains, while only 59 certificates referred to a Root CA in the browser. Manual checks of the underlying domains revealed that non of them could be identified as highly sensitive (political relevant, popular web mailers, anonymization service). Roughly 25% of the certificates where self-signed and differ between Shanghai and TUM-Apr2011. We could not find a good explanation for the differences.

#### 10.4.2.11 Certificate Quality: A Summarizing View

In order to summarize our results, we tried to determine the certificate quality. We group the sites in our data set by Alexa rank and calculate a quality metric for the valid certificates in the group. Each valid certificate can be in one of three groups *good*, *acceptable*, or *poor*. *Good* certificates have no problems in them, i.e. they have correct chains, correct host names, a chain length of at most two, do not use the MD5 signature algorithm, use DSA or RSA keys of at least 1024 bits, and




**(a)** Tue-Nov2009 **(b)** TUM-Apr2011

**Figure 10.15:** Certificate quality in relation to Alexa rank.

have a validity period of maximum 13 months. *Acceptable* certificates are good certificates but may have a longer chain length (up to three) and a validity that is no longer than 25 months. *Poor* certificates are the remainder of the valid certificates that do not match these criteria.

Figure 10.15 reveals the distribution of the quality criteria for Tue-Nov2009 and TUM-Apr2011. One can see that the share of valid certificates is correlated with the rank in the Alexa list: the higher the ranking, the more likely that the domains have a valid certificate. However, we can also see that only little more than regardless of the ranking, more than 50% of the certificates are not valid. Interestingly, the share of *poor* certificates among the valid certificates is higher in the TOP ranked sites compared to the overall sites population. We can see that the overall amount of valid certificates increased over time. The amount of *good* certificates, however, did not increase significantly.

## 10.5 Discussion

The aim of our study was to determine the state of the TLS/SSL deployment and its actual use. We therefore obtained TLS/SSL certificates over a time span of 1.5 years using active measurements, as well as data about TLS/SSL connections from passive measurements. Our data supports a long-standing believe in the security community: the X.509 PKI is to some extend not deployed in the way as it was meant to be deployed. We could find a number of weaknesses in the certificates in our analysis.

One of the most important finding is that the percentage of certificates obtained by users from the top ranking Web sites (top 1 million) without a browser warning is only 18%. Validation failures can be found due to incorrect certification chains (40% of all certificates), as well as to failures to encode the proper host identity into subject or subject alternative name fields. A good finding is that the more popular a site is, the more likely it is to have a valid certificate.

Our study reveals an improvement over time, with more correct certificates deployed over the infrastructure. However, improvement is arriving slowly. Compared with Internet-wide scan results found in the EFF data sets, our Alexa Top 1 million host list analysis shows that more important sites have better certificate deployments than other sites.

However, many certification chains showed multiple errors. Certificate expiration or unknown root certificates have been observed quite often. An interesting observation is the existence of certificates that are shared between different hosts, even for high-profile sites. Improvements such as Extended Validation certificates are not widely used.

Several positive tendencies over time could be observed: With the popularity of a site, the probability that it supports TLS/SSL increases. Furthermore, more popular sites tend to show more valid certificates. We also noted that key lengths tend to be chosen with good lengths, as well as their length increasing over time. Furthermore, discourage signature algorithms tend to be less often used than before, e. g., the use of MD5 seems to be declining. Our passive measurement data also points to the fact that negotiated ciphers tend to be secure with acceptable key lengths.

# Part V

# Conclusion

# 11 Conclusions

In the introduction of this dissertation, we motivated the benefits of traffic analysis systems that are built from commodity hardware. Traffic analysis is known to be an important task that requires many resources in high-speed networks, which was considered the domain of special-purpose hardware in high-speed networks in the past. Recent advances in commodity hardware design allow for handling of network traffic even in high-bandwidth scenarios. However, using commodity hardware with general-purpose operating systems is still considered a difficult task.

In the introduction, we identified several important building blocks of a traffic analysis system, as well as important analysis tasks. Figure 11.1 reiterates the blocks.

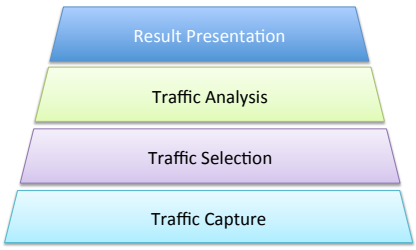



**Figure 11.1:** Building blocks of a traffic analysis system.

The thesis was structured alongside the lower three building blocks and we will now summarize our contributions with respect to the individual blocks.

## 11.1 Traffic Capture

Commodity hardware is usually driven by general-purpose operating systems, which are not optimized for traffic analysis tasks. In Chapter 5, we compared different packet capture solutions of the network stacks of Linux and FreeBSD, including improvements proposed by researchers.

Our findings and contributions can be separated in three groups: First, the evaluation results themselves. In our work, we compared different packet capture techniques that have not been included in a benchmark under the same experiment setups before. Second, we found that a throughout evaluation has to consider multiple application loads, CPU core assignments and other factors. Third, we identified a limitation of the Linux stack that has negative impact on performance.

We propose improvements that mitigate these effects and improve the performance in certain situations.

The evaluation was performed in 2010 with the then-current versions of Linux and FreeBSD. We compared the capture stacks of FreeBSD against the Linux modules PF_PACKET and PF_PFRING. Our evaluation was able to confirm the findings of previous work from Schneider et al. [72] with newer versions of the software: FreeBSD still outperforms standard Linux PF_PACKET under low application load. FreeBSD is especially good when multiple capturing processes are run in parallel. However, we find that FreeBSD always performs worse with traffic analysis applications that have higher application loads. The conclusion of this analysis is that benchmarks of capture stacks should always evaluate the performance with different application loads.

Our comparison between standard Linux capturing with PF_PACKET and the improvement PF_RING confirmed that performance with PF_RING is still better when small packets are to be captured. However, differences are not as big as they where in 2004 or 2006, when the last evaluations were published. Further analyses showed that PF_RING performs better than PF_PACKET if multiple capturing processes are run on the system. Furthermore, we found that the performance of PF_RING is even better than the performance of the FreeBSD capture stack.

Another important finding is the influence of proper scheduling of capture and analysis tasks. We found that default scheduling can result in worse performance compared to capturing with processes that are scheduled manually. These effects can be found on FreeBSD and Linux. A benchmark should therefore also consider performance under various automatic and manual scheduling policies.

During our work, we found a performance bottleneck within the standard Linux capturing module PF_PACKET and PF_RING. As port of our work, we proposed a fix for this problem. This fix greatly improves the performance of PF_PACKET with small packets. After our work was published in [2], PF_RING added the same functionality under the name watermark. Users of PF_RING therefore benefit from our proposed improvement.

Finally, we evaluated Luca Deri's TNAPI driver extension for Linux and found increased performance with all Linux capturing solutions. Best performance can be achieved if TNAPI is combined with PF_RING.

Our findings also show that given certain hardware and high bandwidth and a sufficient computationally complex traffic analysis algorithm, packet loss can still occur. In Chapter 7, we use our findings to configure a multi-core aware capturing system based on the facilities provided by PF_RING. We show how to a well-tuned multi-core aware capturing system can be successfully used in conjunction with appropriate sampling algorithm to maximize the use of the available commodity hardware.

## 11.2 Traffic Selection

Traffic analysis systems can be used for various purposes. Many of them can be grouped into two major classes:

1. Obtain statistical properties on the overall traffic
2. Extract specific information from the traffic

The first class of traffic analysis algorithms aims at providing insight into the overall composition of traffic streams in the network. Relevant statistics can be the number of flows, packets or bytes that have been observed on a given position in the network at a given time. Plenty of research has been conducted on requirements for sampling for this class of analysis. Many algorithms that perform sampling with these applications in mind have been presented in prior research.

The focus of this thesis is on traffic analysis tasks that belong to the second class. All analyses performed throughout this thesis are part of this class. In general, algorithms falling in the class do not aim at providing information about the overall traffic, but more on very specific parts. An example for this type of analysis is security monitoring: The goal of security monitoring is not to generate information on the majority of benign traffic, but on the few packets that carry malicious content. It is therefore crucial for sampling algorithms to ensure a high probability that all or almost all packets of interest are sampled.

In Chapter 3, we studied the requirements of various traffic analysis applications. These included various security monitoring approaches, approaches for traffic classification, or network forensics. The finding of this analysis is that the most interesting parts of the payload can be found at the beginning of communication.

In this thesis, we developed sampling algorithms that work according to this principle. Our analysis showed that the sampling principle provides suitable insight into the traffic for various purposes. In Chapter 8, we showed that we can find most alarms related to botnet traffic at the beginning of the communication. Furthermore, we used the sampling mechanism in Chapter 10 to perform traffic analysis on the X.509 infrastructure by collecting certificates and statistics on TLS/SSL connections.

We presented two algorithms that implement this sampling principle.

### 11.2.1 Static Sampling

The first algorithm was discussed and evaluated throughout Chapter 6. It presented a novel sampling algorithm, which selects packets containing the first $N$ payload bytes of each TCP connection. The sampling algorithm makes use of a simplified TCP connection tracking mechanism. It uses Bloom filters to store connection states and to keep track of the amount of sampled traffic. Through

bloom filters, the algorithm achieves a connection state tracker with constant memory requirements. Moreover, the computational complexity per packet is constant and independent of the observed traffic. Thus, the algorithm can be efficiently implemented in software or hardware to sample traffic at high speed links and pass the selected packets to a subsequent traffic analysis system.

The use of Bloom filters can introduce a source of error due to collisions in hash functions. We evaluated the algorithm using traces from different networks. The evaluation showed that the filters can be dimensioned to provide highly accurate sampling results under normal conditions and acceptable errors under extreme conditions with an unexpectedly high number of connections.

Hence, the Bloom filters can be dimensioned to yield few sampling errors for an expected number of connections. If the observed number of connections exceeds this value, the probability of sampling errors increases only gradually. We were furthermore able to show that the amount of packets that has to be inspected by the following traffic analysis system is dramatically reduced. A sampling limit of a few kilobytes resulted in a reduction of traffic by two orders of magnitude.

### 11.2.2 Dynamic Sampling

One critique of the static sampling approach is that a user is required to pick a static sampling limit $N$. A major concern for security monitoring is the potential for evasion of the sampling process. Another problem is discussed in Chapter 7, where we present several important properties of traffic and traffic analysis systems: Neither is the incoming rate of traffic constant, nor is there a constant packet consumption time for many traffic analysis applications. A good definition for $N$ must therefore consider the worst-case scenario, thus leaving computational resources in all other cases unused.

Chapter 7 addressed this issue by presenting an adaptive load-aware sampling algorithm. Our proposed algorithm overcomes the aforementioned limitations by introducing a dynamic sampling limit. This sampling limit is automatically adapted to match run-time events such as changes in the incoming packet rate or packet consumption rates of the monitoring application. It is chosen such that the monitoring application's utilization of processing power is maximized while random packet loss is minimized. We employ a single indicator, the fill-level of the buffer between the network stack and the traffic analysis application, to determine how the incoming packet rate matches the packet consumption rate of the application. The sampling limit is adopted according to this information.

We evaluated our algorithm in live traffic measurements in a large university network. We found that changes within the traffic and application packet consumption rates can result in significant changes to the sampling limit. Short-term changes in traffic features can influence the processing rate of monitoring applications such that they consume more or less traffic than in the previous time interval. Our dynamic sampling algorithm adapts the sampling limit to those short-term events with the result that the available computing resources are fully utilized. It is capable of being used in

current multi-core aware traffic setups that run multiple instances of monitoring applications, each on a separate core. These capabilities allow including our algorithm into traffic analysis setups that exploit the performance of current multi-core hardware.

## 11.3 Traffic Analysis

### 11.3.1 Worm and Botnet Detection

Chapter 8 analyzed and demonstrated the suitability of our sampling for security monitoring: detection of worm and botnet traffic. We used traffic traces that contain botnet traffic originating from controlled dynamic malware analysis experiments. They have been obtained with a process that installs real malware into a sandbox and records their execution and network traffic. In our evaluation, we could make use of traffic from 93 malware binaries, that we could analyze with signature-based detection methods.

We used the well-known intrusion detection system *Snort* in order to analyze the traffic traces. Snort was modified in a way that allows for recording the position of an alarm within TCP connections and UDP bi-flows. Our analysis revealed that most of the alarms generated by commonly used rule sets can be found within the first few kilobytes of TCP payload.

In a subsequent analysis, we studied the alarm positions in measurements with live traffic. We deployed Snort in a multi-core aware capturing setup on a real network. The deployment ensured that all traffic from that network could be analyzed by the Snort instances. We logged the required number of payload bytes that must be consumed by Snort before the alarm is triggered for each alarm. This evaluation included both TCP and UDP traffic.

In this experiments, we could confirm the findings with the generated traffic traces: TCP alarms are triggered mostly due to payload that resides at the beginning of a TCP connection. UDP alarms require more payload to be triggered to the same extent. However many alarms can be retrieved with little payload as well. We also used BotHunter, a state of the art approach for detecting botnet traffic from related work, to assess the impact of our sampling on its detection result. We also found that BotHunter mostly yields alarms that are generated at the beginning of TCP connections and UDP bi-flows.

### 11.3.2 Caching Benefits for YouTube Traffic

In Chapter 9 we monitored and analyzed traffic between a large operator network and the YouTube video distribution site for over a month. During this time, we observed more than 3.7 million video downloads from PC devices and more than 2.4 million mobile video download connections.

We were able to find good local popularity values for YouTube videos, which result in high caching potential. Several videos where watched quite often in the same video encodings and resolutions within time frames that allows good caching potential. Some user behavior, such as users not fully watching a video, can on the other hand have significant negative impact on caching performance and effectiveness.

In order to yield in good hit rates and therefore benefits, a cache must be carefully configured. It must employ a caching strategy that maximizes content served from the cache and minimizes downloads from the video infrastructure. We analyzed the benefits of simple caching strategies with trace-driven simulations. Our conclusion from these experiments is that simple strategies can result in unnecessary video download traffic and bad caching performance: disk limitations in caches can make the removal of content from the cache necessary. A cache that employs certain replacement strategy removes the wrong content from its internal store, i.e. content that is re-requested by users. Thus, the benefits from the cache are reduced.

We therefore proposed and evaluated a chunked caching and replacement strategy to overcome these problems. In our trace-driven simulation, we where able to show that our proposed strategy outperforms the other approaches. We studied different parameters for our strategies that yield in best performance under caches with different hard disk sizes.

### 11.3.3 Analysis of the TLS/SSL X.509 PKI

TLS and SSL belong to the most important security infrastructures and are used to secure many commercial and non-commercial services. The protocols employ a X.509 certification infrastructure that requires complicated processes in order to ensure proper authentication and the provision of long keys and strong ciphers. Different security researchers have voiced questions about the state of this infrastructure.

In Chapter 10, we applied active and passive measurements to gain a better understanding of the deployment of the PKI, as well as its use. Using active measurements, we obtained X.509 certificates over a time span of one and a half years from different locations. These included university setups at our group's location in Tübingen and Munich, as well as scans from Planet Lab nodes from around the world. In addition, we used a university network monitoring tap to extract TLS/SSL connections over two two-weeks lasting periods. We recorded TLS/SSL connection parameters and collected the certificates that were exchanged during the handshake.

Using this data, we were able to analyze both the certificates and structure of the CAs that issued the certificates. The combination of active and passive data sets provided us with the possibility to gain more insight into the state of the infrastructure, than we could have obtained with only a single type of measurement data. Active data sets where analyzed when questions about the general deployment should be answered. The passive data sets allowed us to gain an understanding on how this infrastructure is actually used.

Our evaluation shows that the actual deployment of X.509 certificates differs from an ideal deployment. We found that the Mozilla Firefox browser only accepts around 18% of the certificates in our data without any warning. The biggest share of the problem, with around 40% of all certificates affected, originates from incorrect certificate chains that prohibit proper validation. Other problems can be found in the encoding of the entity: Host or domain names were often incorrect or missing at all. The problem of incorrect entity encoding was worse in self-signed certificates than in certificates from external CAs.

The evaluation shows improvements throughout our analysis time window of 1.5 years. At the end of our measurement period, more sites supported TLS/SSL compared to the beginning of our study. Key lengths increased in the same time window and weak ciphers where found less often in the end. However, these improvements did only improve the picture to a small degree, with many problems still not being addressed.

## 11.4 Future Directions

The thesis contributed to several areas of network traffic analysis systems, which range from packet capture systems to result reporting. Each of these areas is an active research topic with potential for follow up questions. In the following, we will discuss further ideas that can be examined in future research.

Packet analysis at high speed rates is a very active research topic. In this thesis, we evaluated capture stacks of commodity operating system and provided an evaluation for packet capture systems. Our analysis employed 1 GE hardware because we did not have 10 GE hardware at the time the research was conducted. An obvious extension of our work is to perform the experiments on 10 GE hardware.

Furthermore, researchers propose new architectures and improvements to existing stacks. After our research was conducted, Luca Deri proposed another technique for high-speed packet capture, called *Direct NIC Access* (DNA) [223]. DNA maps a network card's Direct Memory Access (DMA) memory into the user space and implements user space functionality to drive the card. It's design effectively circumvents the operating system, thus freeing up CPU time that is otherwise used for moving packets from the kernel to user land. Other similar mapping technologies exist, for example the Intel's DPDK [224] or UIO-IXGBE [225]. These approaches are praised for good performance, but criticized for the potential instability that circumventing the kernel can impose. Rizzo therefore introduced *netmap*, a packet handling system for FreeBSD and Linux [226]. It provides a framework for integration into the standard operating system kernel. As of today, it is shipped as part of FreeBSD.

These developments show that general-purpose operating systems evolve as new improvements are proposed. The provisioning of a simple automatic framework for continuous benchmarks of packet

capture systems that is available to other researchers could be a task for further research. In our evaluation, we found several important tests that should be included a thorough benchmark. An automatic test framework that allows for simple integration of new approaches can make it easier to provide continuous evaluation of future improvements. Additional tests can be embedded into this test framework as well. We used uniform packet streams and analysis applications that have a uniform distribution of packet-processing times. New tests could be created that use more realistic traffic patterns and more realistic application behavior. Such tests can provide the measurement community with more valuable hints for building and tuning packet capture systems for good performance.

As part of our evaluation of the dynamic sampling algorithm, we assessed the implications of hardware limitations to the chosen sampling limit. Another evaluation could be extended to focus on the implication on the alarms that of Snort or BotHunter compared to a static sampling process. Other research can focus on automatic configuration of the parameters of the sampling algorithm. Our evaluation showed good results with a generic parameter set for various traffic analysis workloads. However, we could also see that the deviation of the sampling limit reached the implementation's boundaries of the kernel level process. The performance of the algorithm can therefore be improved with better parameters.

Research to improve the algorithm can go into two directions: One direction would be to explore the effects of changing the *const* parameter that is the baseline of sampling limit. Additional changes to this baseline in cases that the sampling limit exceeds or undercuts the limits of control process. The second direction can focus on automatic profiling or modeling of network traffic and application behavior. A better understanding of both can help to automatically find parameters.

In our analysis of YouTube downloads, we found good caching potential for the platform's video traffic. However, YouTube is not the only provider for online video download. There exist other download portals for user-generated videos such as Vimeo [227], which could also be analyzed for caching benefits. Another (probably) important example are adult video download portals. A recent study by Tyson et al. at IMC 2013 presented an analysis of content popularity on a large video delivery infrastructure for adult content [228]. They found that video popularity on this platform differed from popularity patterns on video sites such as YouTube. Hence, there could be differences in caching benefits or beneficial caching and replacement strategies. Studies that observe these platforms, both in volume and caching properties, could try to estimate the caching potential for their traffic.

# Bibliography

[1] L. Braun, G. Münz, and G. Carle, "Packet Sampling for Worm and Botnet Detection in TCP Connections," in *12th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2010)*, Osaka, Japan, Apr. 2010.

[2] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, "Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware," in *Proceedings of the 10th Annual Conference on Internet Measurement (IMC '10)*, Melbourne, Australia, Nov. 2010.

[3] R. Holz, L. Braun, N. Kammenhuber, and G. Carle, "The SSL Landscape: A Thorough Analysis of the X.509 PKI Using Active and Passive Measurements," in *Proceedings of the 11th Annual Conference on Internet Measurement (IMC '11)*, Berlin, Germany, Nov. 2011.

[4] L. Braun, A. Klein, G. Carle, H. Reiser, and J. Eisl, "Analyzing Caching Benefits for YouTube Traffic in Edge Networks - A Measurement-Based Evaluation ," in *13th IEEE/IFIP Symposium on Network Operations and Management Symposium (NOMS 2012)*, Maui, Hawaii, Apr. 2012.

[5] L. Braun, M. Volke, J. Schlamp, A. von Bodisco, and G. Carle, "Flow-inspector: A Framework for Visualizing Network Flow Data using Current Web Technologies," in *First IMC Workshop on Internet Visualization (WIV 2012)*, Nov. 2012.

[6] L. Braun, C. Diekmann, N. Kammenhuber, and G. Carle, "Adaptive Load-Aware Sampling for Network Monitoring on Multicore Commodity Hardware," in *Proceedings of the IFIP Networking 2013*, May 2013.

[7] Wikipedia, "Microsoft Office 365," http://en.wikipedia.org/w/index.php?title=Microsoft_Office_365&oldid=572819497, Visited: Dec. 2013.

[8] E. Masanet, A. Shehabi, J. Liang, L. Ramakrishnan, X. H. Ma, V. Hendrix, B. Walker, and P. Mantha, "The Energy Efficiency Potential of Cloud-Based Software: A U.S. Case Study," http://crd.lbl.gov/assets/pubs_presos/ACS/cloud_efficiency_study.pdf, 2013.

[9] Cisco, "Cisco Visual Networking Index: Forecast and Methodology, 2012–2017," http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf, Visited: Dec. 2013.

[10] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of network-based defense mechanisms countering the DoS and DDoS problems," *ACM Computing Surveys (CSUR)*, vol. 39, no. 1, 2007.

[11] "Arbor Networks," http://www.arbornetworks.com/, Visited: Nov. 2103.

[12] "Nmap website," http://nmap.org/, Visited: Dec. 2013.

[13] "Website of the Nessus Project," http://www.tenable.com/products/nessus, Visited: Dec. 2013.

[14] C. Ellison and B. Schneier, "Ten Risks of PKI: What You're not Being Told about Public Key Infrastructure," *Computer Security Journal*, vol. 16, no. 1, pp. 1–7, 2000.

[15] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," in *Proceedings. 11th Symposium on High Performance Interconnects (HOTI '03)*, Stanford, CA, Aug. 2003.

[16] H. Song and J. W. Lockwood, "Efficient Packet Classification for Network Intrusion Detection using FPGA," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays (FPGA 2005)*, Monterey, CA, Feb. 2005.

[17] F. Khan, M. Gokhale, and C.-N. Chuah, "FPGA based Network Traffic Analysis using Traffic Dispersion Patterns," in *Thirteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2005)*, Monterey, California, Feb. 2010.

[18] N. Weaver, V. Paxson, and J. M. Gonzalez, "The Shunt: An FPGA-Based Accelerator for Network Intrusion Prevention," in *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA '07)*, Monterey, CA, Feb. 2007.

[19] V. Paxson, K. Asanovic, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver, "Rethinking hardware support for network analysis and intrusion prevention," in *Proceedings of the 1st USENIX Workshop on Hot Topics in Security (HotSec '06)*, Vancouver, Canada, 2006.

[20] L. Deri, "Improving Passive Packet Capture: Beyond Device Polling," in *Proceedings of the 4th International System Administration and Network Engineering Conference (SANE '04)*, Amsterdam, The Netherlands, Sep. 2004.

[21] M. Roesch, "Snort – Lightweight Intrusion Detection for Networks," in *Proceedings of the 13th USENIX Conference on System Administration (LISA '99)*, Seattle, Washington, Nov. 1999.

[22] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina, "Impact of Packet Sampling on Anomaly Detection Metrics," in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC) 2006*, 2006.

[23] J. Mai, A. Sridharan, C.-N. Chuah, H. Zang, and T. Ye, "Impact of packet sampling on portscan detection," *Journal on Selected Areas of Communication (IEEE)*, vol. 24, no. 12, 2006.

[24] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang, "Is sampled data sufficient for anomaly detection?" in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC) 2006*, Rio de Janeiro, Brazil, Oct. 2006.

[25] B. Trammell and E. Boschi, "Bidirectional Flow Export Using IP Flow Information Export (IPFIX)," RFC 5103 (Proposed Standard), Internet Engineering Task Force, Jan. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5103.txt

[26] B. H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, 1970.

[27] G. Gu, J. Zhang, and W. Lee, "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[28] P. Wendell and M. J. Freedman, "Going Viral: Flash Crowds in an Open CDN," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC 2011)*, Berlin, Germany, 2011.

[29] R. Potharaju and N. Jain, "Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters," in *Proceedings of the 2013 ACM Conference on Internet Measurement Confernece (IMC 2013)*, Barcelona, Spain, Oct. 2013.

[30] "Website of the RANCID project," http://www.shrubbery.net/rancid/, Visited: Dec. 2013.

[31] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC 2010)*, Melbourne, Australia, Nov. 2010.

[32] H. Kim, T. Benson, A. Akella, and N. Feamster, "The Evolution of Network Configuration: A Tale of Two Campuses," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC 2011)*, Berlin, Germany, Nov. 2011.

[33] M. Allman, "Comments on Bufferbloat," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 1, Jan. 2013.

[34] H. Jiang, Z. Liu, Y. Wang, K. Lee, and I. Rhee, "Understanding bufferbloat in cellular networks," in *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular Networks: Operations, Challenges, and Future Design (CellNet 2012)*, Aug. 2012.

[35] Cisco Inc, "Cisco ios ip service level agreements," http://www.cisco.com/en/US/technologies/tk648/tk362/tk920/technologies_white_paper0900aecd8017f8c9.pdf, Visited: Dec. 2013.

[36] N. Hu and P. Steenkiste, "Evaluation and Characterization of Available Bandwidth Probing Techniques," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 6, Aug. 2003.

[37] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with Rocketfuel," in *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2002)*, Pittsburgh, PA, Aug. 2002.

[38] D. Leonard and D. Loguinov, "Demystifying Service Discovery: Implementing an Internet-Wide Scanner," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC 2010)*, Melbourne, Australia, Nov. 2010.

[39] M. Luckie, "Scamper: a scalable and extensible packet prober for active measurement of the internet," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC 2010)*, Melbourne, Australia, Nov. 2010.

[40] M. Tozal and K. Sarac, "Tracenet: an internet topology data collector," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC 2010)*, Melbourne, Australia, Nov. 2010.

[41] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: Illuminating the Edge Network," in *Proceedings of the 10th Annual Conference on Internet Measurement (IMC '10)*, Melbourne, Australia, Nov. 2010.

[42] M. Luckie and B. Stasiewicz, "Measuring Path MTU Discovery Behaviour," in *Proceedings of the 10th Annual Conference on Internet Measurement (IMC '10)*, Melbourne, Australia, Nov. 2010.

[43] E. W. Chan, X. Luo, W. Li, W. W. Fok, and R. K. Chang, "Measurement of Loss Pairs in Network Paths," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC 2010)*, Melbourne, Australia, Nov. 2010.

[44] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig, "Comparing DNS Resolvers in the Wild," in *Proceedings of the 10th Annual Conference on Internet Measurement (IMC '10)*, Melbourne, Australia, Nov. 2010.

[45] B. Trammell, B. Tellenbach, D. Schatzmann, and M. Burkhart, "Peeling Away Timing Error in Netflow Data," in *Proceedings of the 12th Passive and Active Measurement Conference (PAM 2011)*, Atlanta, GA, Mar. 2011.

[46] G. Münz, "Traffic Anomaly Detection and Cause Identification Using Flow-Level Measurements," Ph.D. dissertation, Technische Universität München, Munich.

[47] S. Waldbusser, "Remote Network Monitoring Management Information Base," RFC 2819 (INTERNET STANDARD), Internet Engineering Task Force, May 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2819.txt

[48] ——, "Remote Network Monitoring Management Information Base Version 2," RFC 4502 (Draft Standard), Internet Engineering Task Force, May 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4502.txt

[49] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple Network Management Protocol (SNMP)," RFC 1157 (Historic), Internet Engineering Task Force, May 1990. [Online]. Available: http://www.ietf.org/rfc/rfc1157.txt

[50] L. Cottrell, "Network Monitoring Tools," http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html, Visited: Dec. 2013.

[51] L. Andrey, O. Festor, A. Lahmadi, A. Pras, and J. r. Sch nw lder, "Survey of SNMP performance analysis studies," *International Journal of Network Management*, vol. 19, no. 6, Nov. 2009.

[52] M. Roughan, M. Thorup, and Y. Zhang, "Traffic Engineering with Estimated Traffic Matrices," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement (IMC '03)*, Miami, FL, Oct. 2003.

[53] Q. Zhao, Z. Ge, J. Wang, and J. Xu, "Robust Traffic Matrix Estimation with Imperfect Information: Making Use of Multiple Data Sources," *ACM SIGMETRICS Performance Evaluation Review.*, vol. 34, no. 1, 2006.

[54] Y. Qiao, Z. Hu, and J. Luo, "Efficient Traffic Matrix Estimation for Data Center Networks," in *Proceedings of the IFIP Networking 2013*, New York, NY, May 2013.

[55] B. Claise, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information," RFC 5101 (Proposed Standard), Internet Engineering Task Force, Jan. 2008, obsoleted by RFC 7011. [Online]. Available: http://www.ietf.org/rfc/rfc5101.txt

[56] L. Deri, "Open Source VoIP Traffic Monitoring," http://luca.ntop.org/OpenSourceVoipMonitoring.pdf, Visited: Dec. 2013.

[57] R. T. Lampert, C. Sommer, G. Münz, and F. Dressler, "Vermont - A Versatile Monitoring Toolkit for IPFIX and PSAMP," in *IEEE/IST Workshop on Monitoring, Attack Detection and Mitigation (MonAM2006)*, Tübingen, Germany, Sep. 2006.

[58] J. Kögel, "One-way Delay Measurement based on Flow Data: Quantification and Compensation of Errors by Exporter Profiling," in *Proceedings of the 25th International Conference on Information Networking (ICOIN 2011)*, Chennai, India, Jan. 2011.

[59] B. Claise, A. Johnson, and J. Quittek, "Packet Sampling (PSAMP) Protocol Specifications," RFC 5476 (Proposed Standard), Internet Engineering Task Force, Mar. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5476.txt

[60] B. Claise, "Cisco Systems NetFlow Services Export Version 9," RFC 3954 (Informational), Internet Engineering Task Force, Oct. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3954.txt

[61] "FloMA: Pointers and Software," http://www.switch.ch/network/projects/completed/TF-NGN/floma/software.html, Visited: Dec. 2013.

[62] B. Li, J. Springer, G. Bebis, and M. Hadi Gunes, "A survey of network flow applications," *Journal of Network and Computer Applications*, vol. 36, no. 2, Mar. 2013.

[63] "Ntop Website," http://www.ntop.org, Visited: Dec. 2013.

[64] M. Mellia, A. Carpani, and R. L. Cigno, "TStat: TCP STatistic and analysis tool," in *Second International Workshop on Quality of Service in Multiservice IP Networks (QoSIP2003)*, Milano, Italy, Feb. 2003.

[65] Tstat Homepage, http://tstat.polito.it, Visited: Dec. 2013.

[66] A. Finamore, M. Mellia, M. Meo, M. Munafo, and D. Rossi, "Experiences of Internet Traffic Monitoring with Tstat," *IEEE Network*, vol. 25, no. 3, Apr. 2011.

[67] A. Finamore, M. Mellina, M. M. Munafo, R. Torres, and S. G. Rao, "YouTube Everywhere: Impact of Device and Infrastructure Synergies on User Experience," in *Proceedings of the 11th Annual Conference on Internet Measurement (IMC '11)*, Berlin, Germany, Nov. 2011.

[68] R. Birke, M. Mellia, M. Petracca, and D. Rossi, "Understanding VoIP from Backbone Measurements," in *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM 2007)*, Anchorage, Alaska, May 2007.

[69] "Website of the DIADEM Firewall Project," http://www.diadem-firewall.org/, Visited: Dec. 2013.

[70] P. G. Neumann and P. A. Porras, "Experience with EMERALD to Date." in *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, Apr. 1999.

[71] "CarmentiS- Frühe Warnung im deutschen Internet," in *14. DFN-CERT Workshop*, 2007.

[72] F. Schneider, J. Wallerich, and A. Feldmann, "Packet Capture in 10-Gigabit Ethernet Environments using Contemporary Commodity Hardware," in *Proceedings of the 8th International Conference on Passive and Active Network Measurement (PAM '07)*, Apr. 2007.

[73] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer, "Stateful Intrusion Detection for High-Speed Networks," in *Proceedings of IEEE Symposium on Security and Privacy (S&P 2002)*, Oakland, CA, May 2002.

[74] M. Colajanni and M. Marchetti, "A Parallel Architecture for Stateful Intrusion Detection in High Traffic Networks," in *IEEE/IST Workshop on Monitoring, Attack Detection and Mitigation (MonAM2006)*, Tübingen, Germany, Sep. 2006.

[75] 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 *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (RAID '07)*, Gold Coast, Australia, Sep. 2007.

[76] G. Münz, N. Weber, and G. Carle, "Signature Detection in Sampled Packets," in *Proceedings of Workshop on Monitoring, Attack Detection and Mitigation (MonAM) 2007*, Nov. 2007.

[77] L. Braun and G. Münz, "Netzbasierte Angriffs-und Anomalieerkennung mit TOPAS," in *1. GI FG SIDAR Graduierten-Workshop über Reaktive Sicherheit*, Berlin, Germany, 2006.

[78] G. Münz and G. Carle, "Real-time Analysis of Flow Data for Network Attack Detection," in *10th IFIP/IEEE International Symposium on Integrated Network Management (IM 2007)*, 2007.

[79] "NfSen homepage," http://nfsen.sourceforge.net, Visited: Dec. 2013.

[80] D. Plonka, "Flowscan: A Network Traffic Flow Reporting and Visualization Tool," in *USENIX LISA'00*, New Orleans,LA, Aug. 2012.

[81] T. Taylor, D. Paterson, J. Glanfield, C. Gates, S. Brooks, and J. McHugh, "Flovis: Flow Visualization System," in *Conference For Homeland Security, 2009. CATCH'09. Cybersecurity Applications & Technology*, Mar. 2009.

[82] K. Lakkaraju, W. Yurcik, and A. J. Lee, "NVisionIP: NetFlow Visualizations of System State for Security Situational Awareness," in *Proceedings of VizSEC/DMSEC '04*, Washington, DC, Oct. 2004.

[83] X. Yin, W. Yurcik, and A. Slagell, "VisFlowConnect-IP: An Animated Link Analysis Tool for Visualizing Netflows," in *FLOCON 2005*, Sep. 2005.

[84] Intel Corporation, "An Introduction to the Intel QuickPath Interconnect," http://www.intel.com/technology/quickpath\discretionary{-}{}{}/introduction.pdf, 2009, Visited: Dec. 2013.

[85] J. Mogul and K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-driven Kernel," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 3, 1997.

[86] I. Kim, J. Moon, and H. Yeom, "Timer-based Interrupt Mitigation for High Performance Packet Processing," in *In Proceedings of the 5th International Conference on High-Performance Computing in the Asia-Pacific Region (HPC Asia 2001)*, Gold Coast, Australia, Sep. 2001.

[87] Luigi Rizzo, "Device Polling Support for FreeBSD," in *BSDCon Europe Conference 2001*, 2001.

[88] Linux Foundation, "napi," http://www.linuxfoundation.org/collaborate/workgroups/networking/napi, 2009, Visited: Dec. 2013.

[89] V. Jacobson, C. Leres, and S. McCanne, "libpcap," http://www.tcpdump.org, Visited: Dec. 2013.

[90] Phil Woods, "libpcap MMAP mode on linux," http://public.lanl.gov/cpw/, Visited: Dec. 2013.

[91] F. Fusco and L. Deri, "High Speed Network Traffic Analysis with Commodity Multi-Core Systems," in *Proceedings of the 10th Annual Conference on Internet Measurement (IMC '10)*, Melbourne, Australia, Nov. 2010.

[92] Wikipedia, "Message Signaled Interrupts," http://en.wikipedia.org/w/index.php?title=Message_Signaled_Interrupts&oldid=577626490, 2013, Visited: Dec. 2013.

[93] L. Deri, "nCap: Wire-speed Packet Capture and Transmission," in *Proceeedings of the Workshop on End-to-End Monitoring Techniques and Services,2005 (E2EMon '05)*, Nice, France, May 2005.

[94] G. A. Cascallana and E. M. Lizarrondo, "Collecting Packet Traces at High Speed," in *IEEE/IST Workshop on Monitoring, Attack Detection and Mitigation (MonAM2006)*, Tübingen, Germany, 2006.

[95] N. M. Minnich, "A Packet Capture System for LAN Software Development," in *Proceedings of the 11th Conference on Local Computer Networks*, 1986.

[96] P. D. Amer and L. N. Cassel, "Management of Sampled Real-Time Network Measurements," in *Proceedings of the 14th Conference on Local Computer Networks*, Oct. 1989.

[97] T. Zseby, M. Molina, N. Duffield, S. Niccolini, and F. Raspall, "Sampling and Filtering Techniques for IP Packet Selection," RFC 5475 (Proposed Standard), Internet Engineering Task Force, Mar. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5475.txt

[98] T. Zseby, "Statistical Sampling for Non-Intrusive Measurements in IP Networks," Ph.D. dissertation, Technische Universität Berlin, Berlin, Dec. 2005.

[99] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a Better NetFlow," in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '04)*, Portland, Oregon, 2004.

[100] K. Wang and S. J. Stolfo, "Anomalous Payload-based Network Intrusion Detection," in *Proceedings of the International Symposium on Recent Advances In Intrusion Detection (RAID '04)*, Sophia Antipolis, France, Sep. 2004.

[101] S. Subhabrata, O. Spatscheck, and D. Wang, "Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures," in *Proceedings of the International Conference on World Wide Web, 2004*, New York, NY, May 2004.

[102] Y. J. Won, J.-T. Ju, M.-S. Kim, and J. W. Hong, "A Hybrid Approach for Accurate Application Traffic Identification ," in *Proceedings of IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services (E2EMon '06)*, Vancouver, Canada, Apr. 2006.

[103] L. Bernaille, R. Teixeira, and K. Salamatian, "Early Application Identification," in *Proceedings of the 2006 ACM CoNEXT conference*, Lisboa, Portugal, Dec. 2006.

[104] G. Münz, H. Dai, L. Braun, and G. Carle, "TCP Traffic Classification Using Markov Models," in *Proceedings of Traffic Monitoring and Analysis Workshop (TMA) 2010*, Zurich, Switzerland, Apr. 2010.

[105] G. Münz, S. Heckmüller, L. Braun, and G. Carle, "Improving Markov-based TCP Traffic Classification," in *In Proceedings of the 17th GI/ITG Conference on Communication in Distributed Systems, KiVS 2011*, Kiel, Germany, Mar. 2011.

[106] 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 *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement (IMC '05)*, Berkeley, CA, Oct. 2005.

[107] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, "Enriching Network Security Analysis with Time Travel," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*, Seattle, WA, Aug. 2008.

[108] F. Chang, W. Feng, and K. Li, "Approximate caches for packet classification," in *Proceedings of the 23rd Conference of the IEEE Communications Society (INFOCOMM 2004)*, Hong Kong, China, Mar. 2004.

[109] S. Kong, T. He, X. Shao, and X. Li, "Time-Out bloom Filter: A New Sampling Method for Recording More Flows," in *Proceedings of the International Conference on Information Networking (ICOIN '06)*, Sendai, Japan, Jan. 2006.

[110] S. Kong, X. Shao, C. An, and X. Li, "A Double-Filter Structure Based Scheme for Scalable Port Scan Detection," in *Proceedings of the IEEE International Conference on Communications (ICC '06)*, Istanbul, Turkey, Jun. 2006.

[111] C. Estan, G. Varghese, and M. Fisk, "Bitmap Algorithms for Counting Active Flows on High Speed Links," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement (IMC '03)*, Miami Beach, Fl, Oct. 2003.

[112] J. Sanjuas-Cuxart, P. Barlet-Ros, and J. Solé-Pareta, "Counting Flows over Sliding Windows in High Speed Networks," in *Proceedings of the IFIP NETWORKING 2009*, Aachen, Germany, May 2009.

[113] A. Kumar, J. J. Xu, J. Wang, O. Spatscheck, and E. L. Li, "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement," in *Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004)*, Hong Kong, China, Mar. 2004.

[114] A. Kumar and J. J. Xu, "Sketch Guided Sampling-Using On-Line Estimates of Flow Size for Adaptive Data Collection," in *Proceedings of the Conference of the IEEE Computer and Communications Societies (INFOCOM '2006)*, Barcelona, Spain, Apr. 2006.

[115] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani, "Fast Monitoring of Traffic Subpopulations," in *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement (IMC '08)*, Vouliagmeni, Greece, Oct. 2008.

[116] G. Cormode, S. M. Muthukrishnan, and W. Zhuang, "What's Different: Distributed, Continuous Monitoring of Duplicate-Resilient Aggregates on Data Streams," in *Proceedings. of the IEEE International Conference on Data Engineering (ICDE '06)*, Atlanta, Georgia, USA, Apr. 2006.

[117] B. Whitehead, C.-H. Lung, and P. Rabinovitch, "A TCP Connection Establishment Filter: Symmetric Connection Detection," in *Proceedings of the IEEE International Conference on Communications (ICC '07)*, Glasgow, Scotland, Jun. 2007.

[118] M. Canini, D. Fay, D. J. Miller, A. W. Moore, and R. Bolla, "Per Flow Packet Sampling for High-Speed Network Monitoring," in *Proceedings of the Communication Systems and Networks and Workshops, 2009 (COMSNETS 2009)*, Bangalore, India, Jan. 2009.

[119] R. E. Jurga and M. M. Hulboj, "Packet Sampling for Network Monitoring," Tech. Rep., Dec. 2007.

[120] J. Drobisz and K. J. Christensen, "Adaptive Sampling Methods to Determine Network Traffic Statistics Including the Hurst Parameter," in *Proceedings of the 23rd Annual Conference on Local Computer Networks (LCN 1998)*, Zurich, Switzerland, Oct. 1998.

[121] R. G. Clegg, "A Practical Guide to Measuring the Hurst Parameter," Tech. Rep., Jun. 2006.

[122] K. C. Claffy, G. C. Polyzos, and H.-W. Braun, "Application of Sampling Methodologies to Network Traffic Characterization," in *ACM SIGCOMM Computer Communication Review*, 1993.

[123] B.-Y. Choi, J. Park, and Z.-L. Zhang, "Adaptive Packet Sampling for Flow Volume Measurement," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 3, 2002.

[124] ——, "Adaptive Random Sampling for Load Change Detection," *ACM SIGMETRICS Performance Evaluation Review.*, vol. 30, no. 1, 2002.

[125] J. Zhang, X. Luo, R. Perdisci, G. Gu, W. Lee, and N. Feamster, "Boosting the Scalability of Botnet Detection using Adaptive Traffic Sampling," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*, Mar. 2011.

[126] A. Patcha and J.-M. Park, "An Adaptive Sampling Algorithm with Applications to Denial-of-Service Attack Detection," in *Proceedings of the 15th International Conference on Computer Communications and Networks (ICCCN 2006)*, Arlington, Virginia, Oct. 2006.

[127] E. A. Hernandez, M. C. Chidester, and A. D. George, "Adaptive Sampling for Network Management," *Journal of Network and Systems Management*, vol. 9, no. 4, 2001.

[128] P. Barlet-Ros, G. Iannaccone, J. Sanjuas-Cuxart, D. Amores-Lopez, and J. Solé-Pareta, "Load Shedding in Network Monitoring Applications," in *Proceedings of the USENIX Annual Technical Conference (ATC '07*, Santa Clara, CA, May 2007.

[129] P. Wurzinger, L. Bilge, T. Holz, J. Goebel, C. Kruegel, and E. Kirda, "Automatically Generating Models for Botnet Detection," in *14th European Symposium on Research in Computer Security (ESORICS 2009)*, Saint Malo, France, Sep. 2009.

[130] K. Rieck, G. Schwenk, T. Limmer, T. Holz, and P. Laskov, "Botzilla: Detecting the "Phoning Home" of Malicious Software," in *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*, Sierre, Switzerland, Mar. 2010.

[131] Website of the project *Emerging Threats*, http://www.emergingthreats.net/, Visited: Dec. 2013.

[132] "Website of DARPA Intrusion Detection Evaluation," http://www.ll.mit.edu/IST/ideval/, Visited: Dec. 2013.

[133] R. Sommer and V. Paxson, "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*, Oakland, CA, May 2010.

[134] G. Chatzopoulou, C. Sheng, and M. Faloutsos, "A First Step Towards Understanding Popularity in YouTube," in *INFOCOM IEEE Conference on Computer Communications Workshops*, 2010.

[135] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "Youtube Traffic Characterization: A View From the Edge," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC '07)*, San Diego, CA, Oct. 2007.

[136] M. Zink, K. Suh, Y. Gu, and J. Kurose, "Watch Global, Cache Local: YouTube Network Traffic at a Campus Network-Measurements and Implications," in *Proceedings of the 15th SPIE/ACM Annual Multimedia Computing and Networking Conference (MMCN)*, 2008.

[137] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon, "Analyzing the Video Popularity Characteristics of large-scale User Generated Content Systems," *IEEE/ACM Transactions on Networking (TON)*, vol. 17, no. 5, Oct. 2009.

[138] X. Cheng, C. Dale, and J. Liu, "Statistics and Social Network of YouTube Videos," in *Proceedings of the16th International Workshop on Quality of Service (IWQoS 2008)*, Enschede, The Netherlands, Jun. 2008.

[139] F. Figueiredo, F. Benevenuto, and J. M. Almeida, "The Tube over Time: Characterizing Popularity Growth of Youtube Videos," in *In Proceedings of the 4th ACM Conference on Web Search and Data Mining*, 2011.

[140] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon, "I Tube, You Tube, Everybody Tubes: Analyzing the World's Largest User Generated Content Video System," in *Proceedings of the 7th Conference on Internet Measurements (IMC '07)*, 2007.

[141] R. Torres, A. Finamore, J. R. Kim, M. Mellia, M. Munafo, and S. Rao, "Dissecting Video Server Selection Strategies in the YouTube CDN," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, 2011.

[142] V. K. Adhikari, S. Jain, and Z.-L. Zhang, "YouTube Traffic Dynamics and Its Interplay with a Tier-1 ISP: An ISP Perspective," in *Proceedings of the 10th Annual Conference on Internet Measurement (IMC '10)*, Melbourne, Australia, Nov. 2010.

[143] B. Ager, F. Schneider, J. Kim, and A. Feldmann, "Revisiting Cacheability in Times of User Generated Content," in *INFOCOM IEEE Conference on Computer Communications Workshops*, 2010.

[144] P. Eckersley and J. Burns, "An observatory for the SSLiverse," Talk at Defcon 18. https://www.eff.org/files/DefconSSLiverse.pdf, July 2010, Visited: Dec. 2013.

[145] P. Eckersley and J. Burns, "Is the SSLiverse a safe place?" Talk at 27C3. Slides from https://www.eff.org/files/ccc2010.pdf, 2010, Visited: Dec. 2013.

[146] I. Ristic, "Internet SSL Survey 2010," Talk at BlackHat 2010. Slides from https://media.blackhat.com/bh-us-10/presentations/Ristic/BlackHat-USA-2010-Ristic-Qualys-SSL-Survey-HTTP-Rating-Guide-slides.pdf, 2010, Visited: Dec. 2013.

[147] I. Ristic, "State of SSL," Talk at InfoSec World 2011. Slides from http://blog.ivanristic.com/Qualys_SSL_Labs-State_of_SSL_InfoSec_World_April_2011.pdf, 2011, Visited: Dec. 2013.

[148] Alexa Internet Inc., "Top 1,000,000 sites (updated daily)," http://s3.amazonaws.com/alexa-static/top-1m.csv.zip, 2009–2011, Visited: Dec. 2013.

[149] A. Didebulidze, "Leistungsbewertung und Verbesserung des Packet-Capturings mit PC-Hardware," Diplomarbeit – Technische Universität München, Apr. 2010.

[150] Endace Measurement Systems, http://www.endace.com/, Visited: Dec. 2013.

[151] F. Schneider and J. Wallerich, "Performance Evaluation of Packet Capturing Systems for High-Speed Networks ," in *Proceedings of the 2005 ACM Conference on Emerging Network Experiment and Technology (CoNEXT '05)*, New York, New York, USA, Oct. 2005.

[152] R. Olsson, "pktgen the linux packet generator," http://www.linuxfoundation.org/collaborate/workgroups/networking/pktgen, Visited: Dec. 2013.

[153] tcpdump, http://www.tcpdump.org, Visited: Dec. 2013.

[154] Homepage of the zlib project, http://www.zlib.net/, Visited: Dec. 2013.

[155] C. Satten, "Lossless gigabit remote packet capture with linux," http://staff.washington.edu/corey/gulp/, University of Washington Network Systems, Mar. 2008, Visited: Dec. 2013.

[156] L. Braun, "Verkehrscharakterisierung und Wurmerkennung mit gesampelten Paketen," Diplomarbeit – Universität Tübingen, May 2008.

[157] S. Fide and S. Jenks, "A Survey of String Matching Approaches in Hardware," Dept. of Electrical Engineering and Computer Science, University of California, Irvine, Tech. Rep. TR SPDS 06-01, Mar. 2006.

[158] L. Foschini, A. Thapliyal, L. Cavallaro, C. Kruegel, and G. Vigna, "A Parallel Architecture for Stateful, High-Speed Intrusion Detection," in *ICISS '08: Proceedings of the 4th International Conference on Information Systems Security*, Dec. 2008.

[159] S. Dharmapurikar and V. Paxson, "Robust TCP Stream Reassembly In the Presence of Adversaries," in *Proceedings of the USENIX Security Symposium 2005*, Baltimore, MD, USA, Jul. 2005.

[160] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and its Applications," *Journal of Algorithms*, vol. 55, no. 1, 2005.

[161] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *Journal of Computer and System Science*, vol. 18, no. 2, 1979.

[162] Traffic repository Twente University, http://traces.simpleweb.org, Visited: Dec. 2013.

[163] C. Diekmann, "Adaptive Low-Level Packet Sampling for High-Speed Networks," Interdisciplinary Project – Technische Universität München, Apr. 2011.

[164] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. D. Kolaczyk, and N. Taft, "Structural analysis of network traffic flows," in *Proceedings of the joint international Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2004)*, 2004.

[165] H. Gogl, "Measurement and Characterization of Traffic Streams in High-Speed Wide Area Networks," Ph.D. dissertation, Technische Universität München, Munich.

[166] H. Jiang and C. Dovrolis, "Why is the Internet Traffic Bursty in Short Time Scales?" in *Proceedings of ACM SIGMETRICS 2005*, Banff, Canada, Jun. 2005.

[167] B. Trammell, A. Wagner, and B. Claise, "Flow Aggregation for the IP Flow Information Export (IPFIX) Protocol," RFC 7015 (Proposed Standard), Internet Engineering Task Force, Sep. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7015.txt

[168] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer networks*, vol. 31, no. 23-24, 1999.

[169] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer, "Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection," in *Proceedings of the 15th conference on USENIX Security Symposium (USENIX Security 2006)*, 2006.

[170] "The Bro Network Security Monitor," http://www.bro.org, Visited: Dec. 2013.

[171] D. E. Knuth, *The art of computer programming, volume 2: Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[172] S. Bennett, *A history of control engineering, 1930–1955*. Institute of Electrical Engineers (I.E.E.), 1993.

[173] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On Multi-Gigabit Packet Capturing with Multi–Core Commodity Hardware," in *13th International Conference on Passive and Active Measurement (PAM 2012)*, Vienna, Austria, Mar. 2012.

[174] "Direct NIC Access Website," http://www.ntop.org/products/pf_ring/dna/, Visited: Dec. 2013.

[175] R. Russell, "Unreliable Guide To Hacking The Linux Kernel," https://www.kernel.org/doc/htmldocs/kernel-hacking/index.html, 2005, Visited: Dec. 2013.

[176] J. Verdu, J. Garcia, M. Nemirovsky, and M. Valero, "Architectural Impact of Stateful Networking Applications," in *Proceedings of the 2005 Symposium on Architectures for Networking and Communications Systems (ANCS 2005)*, Princeton, NJ, Oct. 2005.

[177] J. Goebel and T. Holz, "Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation," in *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets (HotBots 07)*, 2007.

[178] J. A. Morales, A. Al-Bataineh, S. Xu, and R. Sandhu, "Analyzing DNS activities of bot processes," in *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, 2009.

[179] J. Nazario and T. Holz, "As the net churns: Fast-flux botnet observations," in *4th International Conference onMalicious and Unwanted Software (MALWARE 2008)*, 2008.

[180] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi, "EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis," in *Proceedings of the Network and Distributed System Security Symposium (NDSS 2011)*, 2011.

[181] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection." in *Proceedings of 17th USENIX Security Symposium*, San Jose, CA, Jul. 2008.

[182] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A Survey on Automated Dynamic Malware-Analysis Techniques and Tools," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, 2012.

[183] C. Gorecki, F. C. Freiling, M. Kührer, and T. Holz, "TRUMANBOX: improving dynamic malware analysis by emulating the internet," in *Stabilization, Safety, and Security of Distributed Systems*, 2011.

[184] E. Raftopoulos and X. Dimitropoulos, "Detecting, validating and characterizing computer infections in the wild," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement (IMC 2011)*, Berlin, Germany, Nov. 2011.

[185] "BotHunter Website," http://www.bothunter.net/, Visited: Dec. 2013.

[186] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian, "Internet Inter-Domain Traffic," in *Proceedings of the ACM SIGCOMM 2010 Conference on SIGCOMM*, New Delhi, India, Aug. 2010.

[187] G. Maier, A. Feldmann, V. Paxson, and M. Allman, "On Dominant Characteristics of Residential Broadband Internet Traffic," in *Proceedings of the 9th Annual Conference on Internet Measurement (IMC '09)*, Chicago, Illinois, Nov. 2009.

[188] R. Zhou, S. Khemmarat, and L. Gao, "The Impact of YouTube Recommendation System on Video Views," in *Proceedings of the 10th Annual Conference on Internet Measurement (IMC '10)*, Melbourne, Australia, Nov. 2010.

[189] J. Postel and J. Reynolds, "Instructions to RFC Authors," RFC 2223 (Informational), Internet Engineering Task Force, Oct. 1997, updated by RFCs 5741, 6949. [Online]. Available: http://www.ietf.org/rfc/rfc2223.txt

[190] E. Rescorla and B. Korver, "Guidelines for Writing RFC Text on Security Considerations," RFC 3552 (Best Current Practice), Internet Engineering Task Force, Jul. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3552.txt

[191] S. Kent and R. Atkinson, "IP Authentication Header," RFC 2402 (Proposed Standard), Internet Engineering Task Force, Nov. 1998, obsoleted by RFCs 4302, 4305. [Online]. Available: http://www.ietf.org/rfc/rfc2402.txt

[192] ——, "IP Encapsulating Security Payload (ESP)," RFC 2406 (Proposed Standard), Internet Engineering Task Force, Nov. 1998, obsoleted by RFCs 4303, 4305. [Online]. Available: http://www.ietf.org/rfc/rfc2406.txt

[193] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246 (Proposed Standard), Internet Engineering Task Force, Jan. 1999, obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc2246.txt

[194] A. Freier, P. Karlton, and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0," RFC 6101 (Historic), Internet Engineering Task Force, Aug. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6101.txt

[195] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[196] C. Adams and S. Farrell, "Internet X.509 Public Key Infrastructure Certificate Management Protocols," RFC 2510 (Proposed Standard), Internet Engineering Task Force, Mar. 1999, obsoleted by RFC 4210. [Online]. Available: http://www.ietf.org/rfc/rfc2510.txt

[197] C. Adams, S. Farrell, T. Kause, and T. Mononen, "Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP)," RFC 4210 (Proposed Standard), Internet Engineering Task Force, Sep. 2005, updated by RFC 6712. [Online]. Available: http://www.ietf.org/rfc/rfc4210.txt

[198] "Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates, v.1.0," https://cabforum.org/wp-content/uploads/Baseline_Requirements_V1_1_6.pdf, Jul. 2013, Visited: Dec. 2013.

[199] "Mozilla CA Certificate Inclusion Policy (Version 2.2)," http://www.mozilla.org/projects/security/certs/policy/InclusionPolicy.html, Visited: Dec. 2013.

[200] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280 (Proposed Standard), Internet Engineering Task Force, May 2008, updated by RFC 6818. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt

[201] J. Appelbaum, "Detecting certificate authority compromises and web browser collusion," https://blog.torproject.org/blog/detecting-certificate-authority-compromises-and-web-browser-collusion, 2011, Visited: Dec. 2013.

[202] M. S. Blog, "DigiNotar removal follow up," https://blog.mozilla.com/security/2011/09/02/diginotar-removal-follow-up/, 2011, Visited: Dec. 2013.

[203] C. Herley, "So long, and no thanks for the externalities: the rational rejection of security advice by users," in *Proc. 2009 Workshop on New security paradigms*. New York, NY, USA: ACM, 2009, pp. 133–144.

[204] P. Gutmann, "PKI: It's not dead, just resting," *IEEE Computer*, vol. 35, no. 8, pp. 41–49, August 2002.

[205] Data sets of active scans, http://pki.net.in.tum.de, 2011, Visited: Dec. 2013.

[206] A. Croll and S. Power, *Complete Web Monitoring*. O'Reilly Media, 2009.

[207] G. F. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. USA: Insecure, 2009.

[208] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer, "Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection," in *Proc. USENIX Security Symposium*, Apr. 2006.

[209] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright, "Transport Layer Security (TLS) Extensions," RFC 3546 (Proposed Standard), Internet Engineering Task Force, Jun. 2003, obsoleted by RFC 4366. [Online]. Available: http://www.ietf.org/rfc/rfc3546.txt

[210] Planet Lab, "Planet Lab Website," https://www.planet-lab.org, 2011, Visited: Dec. 2013.

[211] The International Grid Trust Federation, "Igtf website," http://www.igtf.net/, 2011, Visited: Dec. 2013.

[212] E. Butler, "Firesheep," http://codebutler.com/firesheep/, Oct. 2010, Visited: Dec. 2013.

[213] I. Security, "Facebook boosts security with ssl encryption," http://www.informationweek.com/security/application-security/facebook-boosts-security-with-ssl-encryp/229100364, Jan. 2011, Visited: Dec. 2013.

[214] T. Blog, "Making Twitter more secure: HTTPS," https://blog.twitter.com/2011/making-twitter-more-secure-https, Mar. 2011, Visited: Dec. 2013.

[215] B. Schneier, *Applied Cryptography, Second Edition*. John Wiley & Sons, 1996.

[216] CA/Browser Forum, "EV SSL Certificate Guidelines Version 1.3," https://cabforum.org/wp-content/uploads/Guidelines_v1_4_3.pdf, 2010, Visited: Dec. 2013.

[217] M. Stevens, A. Lenstra, and B. de Weger, "Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities," in *Advances in Cryptology – EUROCRYPT 2007*, ser. LNCS. Springer Berlin / Heidelberg, 2007, vol. 4515, pp. 1–22.

[218] NIST, “Approved Algorithms,” http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html, 2006, Visited: Dec. 2013.

[219] A. Sotirov, M. Stevens, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger, “MD5 considered harmful today,” http://dl.packetstormsecurity.net/papers/attack/md5-considered-harmful.pdf, 2008, Visited: Dec. 2013.

[220] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann, “Factorization of a 768-bit rsa modulus,” in *Advances in Cryptology – CRYPTO 2010*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6223, pp. 333–350.

[221] NIST, “Special Publications (800 Series),” http://csrc.nist.gov/publications/PubsSPs.html, 2011, Visited: Dec. 2013.

[222] Debian Project, “Debian Security Advisory: DSA-1571-1 openssl – predictable random number generator,” http://www.debian.org/security/2008/dsa-1571, 2008, Visited: Dec. 2013.

[223] “Direct NIC Access - Gigabit and 10 Gigabit Ethernet Line-Rate Packet Capture and Injection,” http://www.ntop.org/products/pf_ring/dna/, Visited: Dec. 2013.

[224] Intel, “Intel data plane development kit,” http://edc.intel.com/Link.aspx?id=5378, Visited: Dec. 2013.

[225] “UIO-IXGBE Project Page,” https://opensource.qualcomm.com/wiki/UIO-IXGBE, Visited: Dec. 2013.

[226] L. Rizzo, “netmap: a novel framework for fast packet I/O,” in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC 2012)*, Boston, MA, Jun. 2012.

[227] “Vimeo,” http://www.vimeo.com, Visited: Dec. 2013.

[228] G. Tyson, Y. Elkhatib, and N. Sastry, “Demystifying Porn 2.0: A Look into a Major Adult Video Streaming Website,” in *Porceedings of ACM/SIGCOMM Internet Measurement Conference (IMC 2013)*, Barcelona, Spain, Oct. 2013.

ISBN 3-937201-42-4

ISSN 1868-2634 (print)
ISSN 1868-2642 (electronic)