HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Felix Erlacher

Efficient Intrusion Detection in High-Speed Networks

Dissertation

April 2019

Please cite as:
Felix Erlacher, “Efficient Intrusion Detection in High-Speed Networks,” PhD Thesis (Dissertation), Heinz Nixdorf Institute, Paderborn University, Germany, June 2019.

Distributed Embedded Systems (CCS Labs)
Heinz Nixdorf Institute, Paderborn University, Germany

Fürstenallee 11 · 33102 Paderborn · Germany

http://www.ccs-labs.org/

Efficient Intrusion Detection in High-Speed Networks

Dissertation
zur Erlangung des Grades

DOKTOR DER NATURWISSENSCHAFTEN

vorgelegt von

Felix Erlacher

geb. am 14. Juli 1981
in Meran, Italien

angefertigt in der Fachgruppe

Distributed Embedded Systems
(CCS Labs)

Heinz Nixdorf Institut
Universität Paderborn

Betreuer:	**Prof. Dr.-Ing. Falko Dressler**
Gutachter:	**Prof. Dr.-Ing. Falko Dressler** **Prof. Dr.-Ing. Felix Freiling**
Tag der Abgabe:	**26. April 2019**
Tag der Promotion:	**27. Juni 2019**

Abstract

To keep today's computer networks up and running, it is paramount to detect all attacks and malicious activities contained in the network traffic. This makes network intrusion detection an integral part of every IT security strategy. In this PhD thesis we study the problem of intrusion detection in high-speed networks. To achieve sufficient accuracy, state-of-the-art Network Intrusion Detection Systems (NIDS) apply performance intensive procedures like Deep Packet Inspection (DPI)-methods on network packets and, thus, can not cope with the traffic rates in high-throughput networks. The fact that high-throughput connections are nowadays widespread even in smaller corporate or campus networks, stresses for more efficient detection approaches. This thesis proposes novel methods for efficient intrusion detection in such scenarios. In order to get an understanding of today's threat landscape, we give an overview of the attacks which arose with the introduction of the so called Web 2.0. We analyze current mitigation techniques and point out open research problems. Further, we present an approach which increases the efficiency of anomaly-based NIDS by combining multiple anomaly detection algorithms on a single computer. Our novel load allocation scheme mitigates random packet drops caused by the high performance-demand of the combined algorithms. To increase the network throughput performance of network monitoring appliances in general and NIDS in particular, we propose two methods for preprocessing HTTP traffic before analysis. We show that both approaches significantly reduce the data portion to be analyzed while retaining the relevant parts for intrusion detection. Then we present our novel signature-based NIDS called FIXIDS, which takes as input HTTP-enriched IPFIX Flows. By applying HTTP-related signatures from the widely used NIDS Snort, it guarantees that thousands of up-to-date and community validated attack descriptions are available. Results show that FIXIDS is able to analyze the HTTP-portion of typical internet traffic even at rates of more than 9.5 Gbit/s. In the final contribution we propose a malicious HTTP traffic generator for NIDS evaluation called GENESIDS. It uses Snort signatures as attack descriptions. The evaluation shows that GENESIDS reliably generates a variety of more than 8000 different attacks. Summarizing, we strongly believe that the above contributions significantly increase the efficiency of NIDS in modern high-speed networks.

Kurzfassung

Um heutige Computer Netzwerke in Betrieb zu halten, ist es unumgänglich alle Angriffe und bösartigen Aktivitäten im Netzwerkverkehr zu entdecken. Diese Anforderung macht Angriffserkennung zu einem integralen Bestandteil jeder IT-Sicherheitsstrategie. In dieser PhD Arbeit beschäftigen wir uns mit dem Problem der Angriffserkennung in Hochgeschwindigkeitsnetzwerken. Um eine zufriedenstellende Genauigkeit bei der Angriffserkennung zu erreichen, greifen moderne Angriffserkennungssysteme auf leistungsintensive Methoden wie Deep Packet Inspection (DPI) zurück und können deshalb nicht mehr mit dem Verkehrsaufkommen in Hochgeschwindigkeitsnetzen mithalten. Die Tatsache, dass Hochgeschwindigkeitsverbindungen heutzutage sogar in kleineren Firmen- und Campusnetzwerken weit verbreitet sind, unterstreicht die Wichtigkeit, welche die Entwicklung von effizienten Angriffserkennungssystemen hat. In dieser Arbeit schlagen wir neuartige Methoden für effiziente Angriffserkennung vor. Wir geben einen Überblick über die aktuelle Bedrohungslandschaft im Internet. Hierbei konzentrieren wir uns speziell auf Bedrohungen, welche mit der Einführung des sogenannten Web 2.0 entstanden. Wir untersuchen die aktuellen Gegenmaßnahmen und zeigen offene Problemstellungen auf. Dann zeigen wir, wie man die Effizienz von Anomalie-basierten Angriffserkennungssystemen erhöhen kann, indem wir mehrere Anomalieerkennungs-Algorithmen auf einer Maschine kombinieren. Durch die hohe Leistungsanforderung der kombinierten Algorithmen können zufällige Paketverluste auftreten, diese mildern wir durch eine neuartigen Herangehensweise ab. Um den Datendurchsatz von Netzwerk-Monitoring Vorrichtungen im Allgemeinen und Angriffserkennungssystemen im Speziellen zu erhöhen, schlagen wir zwei Methoden zur Vorverarbeitung von HTTP Verkehr vor. Wir zeigen, dass beide Methoden die Menge der zu analysierenden Daten signifikant reduzieren und dabei die für die Angriffserkennung interessanten Daten erhalten. Dann präsentieren wir unser neuartiges Signatur-basiertes Angriffserkennungssystem FIXIDS, welches HTTP-haltige IPFIX Flows analysiert. Indem wir HTTP-basierte Angriffssignaturen des weit verbreiteten Angriffserkennungssystems Snort verwenden, ist garantiert, dass tausende aktuelle und von der Community gepflegte Angriffssignaturen zur Verfügung stehen. Die Evaluation zeigt, dass FIXIDS den HTTP Teil von typischem Internet Verkehr sogar mit Raten von mehr

als 9.5 Gbit/s verlustfrei analysieren kann. Im abschließenden Beitrag dieser Arbeit schlagen wir einen Verkehrsgenerator für bösartigen Verkehr namens GENESIDS vor. Hierbei verwenden wir Snort Signaturen als Angriffsbeschreibungen. Die Evaluation zeigt, dass GENESIDS verlässlich abwechslungsreichen Verkehr mit mehr als 8000 verschiedenen Angriffen generiert. Zusammenfassend sind wir der Überzeugung, dass die oben genannten Beiträge die Effizienz von Angriffserkennungssystemen in modernen Hochgeschwindigkeitsnetzwerken signifikant steigern.

Contents

Chapter 1

Introduction

TODAY'S world can not be imagined without computers and information systems. According to the International Telecommunication Union (ITU) [1], in 2016, more than 80 % of households in developed countries had access to the internet. Considering that the same report states, that developed countries have almost 100 active mobile-broadband subscriptions per 100 inhabitants, we can determine that the internet is a crucial part of our everyday life.[1] And these numbers are steadily rising.

The pervasiveness of the internet has a major impact on how we organize our life: Almost all channels of communication have moved to the internet. We use the internet on a daily basis to gather information about weather, news or traffic. We conduct our money transfers via online banking and manage our citizen duties over e-government applications. All this results in private persons, companies, and whole societies being directly dependent on the internet.

However, while it is desirable to increase the ubiquitousness of the internet, a number of challenges arise: Firstly, the infrastructure has to keep up with the rising demands. While a couple of years ago, one fiber connection provided enough bandwidth to supply a small town, nowadays the trend goes towards one fiber connection per home. Trevisan et al. [2] show in their longitudinal study of Internet Service Provider (ISP) traffic, that the average daily download volume of a single internet subscriber has increased from an average of 300 MByte in 2013, to 700 MByte in 2017. This implies that the network throughput, that ISPs have to handle in their backbone network links, rises continuously.

Secondly, attacks on computer systems have become everyday reality. The U.S. Federal Bureau of Investigation (FBI) runs the Internet Crime Complaint Center (IC3) to collect data about cybercrime incidents. In the 2017 annual report[2] they state a total reported loss of 5.52 billion US dollars. Symantec, one of the biggest cybersecurity software companies, states in its "Internet Security Threat Report" for 2017 [3], that the percentage of malicious web traffic in its monitored traffic grew from 5 % to 7.8 % compared to 2016.

Summarizing, the focus of criminals on the internet is simply a question of costs and benefits. Adding to this, that criminals and their victims, most times, reside in areas with completely different jurisdictions, it seems obvious that cybercrime is increasing.

There are numerous things that threaten the machines and users, which comprise a computer network: It starts with attacks on the operational state of computer systems and computer networks; here Distributed Denial of Service (DDOS) attacks are among the most common threats [4], [5]. But also sophisticated attacks on

[1] The pervasiveness of the internet is considerably lower in developing countries: 50 active mobile-broadband subscriptions per 100 inhabitants. But the trend in developing countries shows a much steeper increase in the last years compared to developed countries.

[2] `https://www.ic3.gov/media/annualreport/2017_IC3Report.pdf`

known vulnerabilities or weaknesses of the targeted system can be very successful and, more importantly, may remain undetected because of the stealthy nature of the attack.

Especially in recent years, the number of computer systems that have been victims of malware has risen significantly. The type of malware gaining most media attention are so called “cryptolockers” or “ransomware” [6], where attackers restrict access to the victims’ computer resources (e.g., by encrypting the hard drive) and then ask for some form of payment to remove the restriction. Chapter 3 provides a more in-depth elaboration on current internet threats.

Securing computer systems starts by using software and hardware, that is provided with updates and patches to vulnerabilities that are discovered after the affected software has been shipped and installed. But most important is to install updates and patches for such flaws as soon as they are available. The reason for this, is that almost all significant intrusions and malware attacks exploit known vulnerabilities to gain access to attacked computer systems.

Additional countermeasures include so called Anti-Virus software (AV software). While it was initially developed with the aim to detect viruses, modern AV software prevent, detect and remove all sorts of malware, and some AV software suites even protect against browser-based attacks (cf. Chapter 3).

But the most widely used antidote to network attacks, especially in corporate and campus environments, are Network Intrusion Detection Systems (NIDS), which we describe in detail in the next section.

1.1 Intrusion Detection Systems

Intrusion Detection Systems (IDS) in general, are appliances that monitor a network or a computer system for *intrusions* [7]. The term intrusion detection, as used in IDS, can have different meanings: it might describe the detection of an actual intrusion, where an attacker illegitimately tries to gain access to a corporate network, but includes also the detection of malware and viruses or the enforcement of corporate policies (e.g., no sharing of pdf documents via e-mail). Depending on the system and the configuration, an IDS will trigger an alarm or event if it detects what it defines as an intrusion.

Traditionally, IDS are categorized according to the data they are analyzing: Host-based Intrusion Detection Systems (HIDS) are placed on the host of interest and typically monitor Operating System (OS) operations and log files (e.g., *OSSEC* [8]). On the other hand we have NIDS, which analyze network traffic data. For this thesis of interest is the latter type.

According to established taxonomies [9]–[11], NIDS can be put in two categories depending on the used detection method:

- The first category are anomaly-based NIDS [11], [12]. They use behavior-based techniques, implemented in Anomaly Detection Algorithms (ADAs) by defining a model of normal network behavior, and then detecting deviations to this model. Usually, this detection analysis is done, by applying an application-specific ADA on a data set that should be examined for anomalies. The model of normal traffic is built during a so called training phase. Depending on the ADA, this model is built manually or automatically. A manually build model of normal traffic usually consists of traffic characteristics and parameters that can be specified by the user. Automatically generated traffic models, on the other hand, are produced by an ADA analyzing benign or normal traffic and generating the model using characteristics gathered during this analysis. Abundant research effort has been put into using machine learning techniques for the automatic generation of such traffic models [13].

  Anomalies identified during the detection phase, can be classified by type, severity or any other relevant property. Most times, though, anomalies are marked with attributes like floating-point scores or numeric class values. These scores and values can then be compared to different thresholds, to be able to sort them by severity.

  Examples of renowned, anomaly-based NIDS include *SPADE* [14], *PAYL* [15] or *NICE* [16]. The performance of each anomaly-based NIDS depends heavily on the mechanics of the used ADA. ADAs that rely solely on packet header information usually have a higher packet throughput performance, but come at the cost of a significantly lower detection accuracy compared to ADAs which consider also the payload. Payload-based systems, on the other hand, show a higher detection rate at the cost of a much lower data throughput.

- The second category of NIDS are signature-based systems. They use a precise definition of known attacks and match bypassing network traffic against this definition. This implies that a database with definitions of observed attacks has been established in advance. Such attack definitions are called signatures or rules. For the rest of this thesis we will use both notations interchangeably. To narrow down the number of packets a rule has to be applied to, the attack definitions are usually combined with a network address range or similar traffic filters.

  Signature-based NIDS incorporate a detection engine, which applies the rules to bypassing network data. Such rules typically contain patterns that are matched against the payload of the received packets. These patterns range

from selected bytes to complex Regular Expressions (RegExes), matching not only individual packets, but payload in a stream of packets. For example, the password cracker Brutus can be used to guess passwords of web applications. For every password guess it issues a Hypertext Transfer Protocol (HTTP) request with the following ASCII pattern in the user agent field of the HTTP header: "Mozilla/3.0 (Compatible);Brutus/AET". A simple and effective signature, written to detect such an intrusion attempt, will contain this pattern. If a NIDS uses such a signature and matches this pattern in the bypassing network traffic, it will trigger an alert.[3]

The main steps of a signature-based NIDS are the following: First, the traffic goes through a decoding phase, where the structure of incoming frames is defined (e.g., the start and end of the single protocol header fields).

Then the traffic is processed by a predefined set of preprocessors. Here, packets are reassembled (e.g., fragmented TCP packets) and checked for validity (e.g., TCP checksum). The applied preprocessing steps depend mainly on the applied intrusion detection methods. If the signatures allow the definition of patterns in HTTP fields, for example, then the preprocessing of packets will contain an HTTP parsing step.

Finally, the preprocessed data is used in the detection phase. Here, all signature patterns are applied to the incoming data. This is the most performance intensive step of a signature-based NIDS. Thus, it is important that the data is preprocessed properly so that the pattern-matching operations can be applied in an efficient way. For example, if input data for the pattern-matching process already contains the parsed HTTP fields, then the pattern matching can be performed much more efficiently compared to the matching of HTTP patterns on raw application-layer data.

The performance of signature-based NIDS mainly depends on the number of applied signatures [17]. Thus, in practical applications, the rule-set is adapted for the domain-specific use case. Nevertheless, for a comprehensive attack coverage, a high number of rules remain, and further reducing the number of rules would elevate the risk of not detecting a possible intrusion. Thus, such systems can only cope with a relatively low network throughput rate.

By far the most widespread rule-based NIDS is Snort [18]. Snort is Free and Open Source Software (FOSS) and maintained by Sourcefire, a company owned by Cisco.[4] Snort signatures not only describe malicious activity like network attacks or malware, but also scanning and fingerprinting attempts or attacks on the browser engine, to name a few.

[3] The signature with the unique SID identification number 26558, for the widely used NIDS Snort, contains exactly this pattern to detect intrusion attempts performed with the help of Brutus.

[4] `https://www.cisco.com`

Because of its popularity, Snort also has the biggest source of community-driven and up-to-date rule databases. So it does not come as a surprise, that many other signature-based NIDS use Snort rules and / or the Snort rule syntax as input for their attack definitions.

There are also other noteworthy signature-based NIDS like *Suricata*[5] or *Zeek*[6] [19] (formerly known as *Bro*). But they do not own the same widespread popularity as Snort.

Some NIDS, including Snort, not only offer intrusion detection capabilities, but can also be run as Intrusion Prevention System (IPS). In such a configuration, these systems react on a generated event, typically trying to prevent or abort the detected attack attempt, e.g., by resetting the connection or notifying a firewall system to block further connection attempts.

Regardless of the detection method, NIDS will introduce detection errors. An anomaly-based NIDS might falsely classify normal traffic as an anomaly, and a signature-based NIDS might classify benign traffic as an attack attempt. Generally speaking, there are four classes of output types of NIDS:

- **False positive:** If a system falsely triggers an alarm for an input that is benign – also called false alarm.
- **False negative:** If a system falsely classifies an input as benign, although it should be classified as an intrusion.
- **True negative:** If a system correctly classifies an input as benign.
- **True positive:** If a system correctly classifies an input as an intrusion.

Because both, signature-based and anomaly-based systems, have their own merits and drawbacks, there have been efforts to combine both approaches into a hybrid IDS (e.g., [20], [21]).

Generally, both categories of NIDS try to find a balance between high network throughput performance and high detection accuracy. Analyzing the application-layer part of a packet or a stream (also called Deep Packet Inspection (DPI)) promises higher detection accuracy. This, however, has a critical impact on the network throughput performance, because the system has to spent more time on the analysis of single packets or streams. The challenge is to adapt the accuracy and attack coverage to an acceptable level, while making sure that the NIDS can still analyze the entire network traffic.

If the application scenario requires a high detection accuracy, the throughput performance might deteriorate to a point where not all network traffic can be analyzed

[5] `https://suricata-ids.org/`
[6] `https://zeek.org`

anymore. If the analysis rate is continuously lower than the incoming packet rate, then the traffic buffer will overflow, which finally leads to uncontrolled packet drops. For an intrusion detection environment this is a situation that should be avoided at all costs, because intrusions contained in dropped packets can not be reported.

To avoid such scenarios, there exist multiple approaches which try to reduce the amount of data that needs to be analyzed. A widely used approach, not only for intrusion detection, but for network monitoring in general, is Flow-based network traffic analysis. Hereby, the incoming network traffic is aggregated to so called Flow Records and these Flow Records are then used for analysis. Such Flow Records contain usually only statistics or samples from the original traffic data and, thus, the data volume is reduced drastically. The state-of-the-art standard for Flow-based data is the Internet Protocol Flow Information Export (IPFIX) protocol. Because of the lack of payload-based information in Flow data, until now, only anomaly-based NIDS use Flow Records for intrusion detection.

Summarizing, both categories of NIDS have their advantages and drawbacks. Anomaly-based systems detect anomalous events and, thus, can also detect previously unknown attacks. Signature-based systems need a precise definition of the attack, therefore they can not detect unknown attacks. But because not every anomaly is an attack or even a noteworthy event, anomaly-based systems suffer from a relatively high false positive rate. Signature-based systems, on the other hand, benefit from a precise description of attacks and, thus, have a substantially lower false positive rate.

Typically, signature-based NIDS have a lower network traffic throughput compared to anomaly-based NIDS, because they use DPI-based analysis, and the amount of rules applied to the bypassing traffic is usually relatively high. However, to work as expected, both types of NIDS need to be configured and adapted according to the expected network traffic.

Finally, it depends on the user to decide which NIDS to use for a specific application scenario and to make sure to adapt it to the expected network traffic to achieve the desired results.

Apart from Chapter 4, this thesis deals mostly with signature-based NIDS.

1.2 Research Questions

In the following we outline the research questions that this thesis focuses on. The main challenge that we will solve, is the relatively low network throughput of current NIDS, which can not keep up with modern high-speed networks. The are two requirements to the solutions: One is to offer solutions running on off-the-shelve hardware to make sure that they can be applied in a wide range of application

scenarios. The second is to publish the developed prototypes and software under a FOSS license to make sure that everybody can study, adapt and improve what we have developed.

Anomaly-based NIDS can generally cope with higher network throughput rates compared to signature-based systems, because the used algorithms are less performance intensive than the ones used in signature-based systems. The downside of anomaly-based systems is that their detection accuracy suffers from a high number of false positives. Combining multiple ADAs promises increased detection accuracy, but will also decrease the network throughput performance, because the high computational load of multiple algorithms is put on a single system. Thus, the first research question is how can we combine multiple ADAs on a single machine, mitigating the negative impact of the high computational load, caused by multiple ADAs?

With the second research question we try to tackle the problem of high traffic volume in intrusion detection from a different perspective. HTTP is the most used application-layer protocol on the internet. In addition, the biggest share of intrusions is found in the HTTP part of network traffic. Therefore, modern NIDS put a lot of effort in decoding and preprocessing HTTP traffic for efficient intrusion detection. But only a very small part of the overall HTTP traffic contains intrusions. Therefore, the second research question is how to reduce the amount and aggregate the interesting parts of HTTP traffic for network monitoring and intrusion detection? Our focus hereby, is to reduce the traffic volume for intrusion detection analysis without reducing the detection accuracy.

A common approach for reducing the amount of analyzed traffic is Flow-based intrusion detection. But, until now, there exist only anomaly-based NIDS focusing on Flow records. The reason for this is the complete lack of application-layer information in Flow records. Accordingly, the third research question is the following: Can we use HTTP-enriched IPFIX Flows for efficient intrusion detection?

Because of the experiences made during the development of own NIDS prototypes, and as a result of numerous discussions with fellow researchers, we realized that there is a lack of publicly available malicious traffic traces, which contain a representable and comprehensive set of attacks for NIDS evaluation. This is mainly because researchers use own, handcrafted traffic or traffic which is captured from a university uplink or the like, which entails that the traffic contains privacy sensitive data and can thus, not be published. This makes representable and comprehensive NIDS evaluation and its replication almost impossible. Thus, the fourth research question is how can we automatically generate malicious test traffic for NIDS, which includes a representable and comprehensive set of attacks?

1.3 Thesis Organization and Contribution

In this section we present the structure of this thesis and briefly summarize the content and contributions of the single chapters.

In **Chapter 2** we give an introduction to fundamental concepts that this thesis is based on. We also introduce related work to the topics of this thesis and elaborate especially on approaches to increase the network throughput of NIDS. Then, we introduce the concept of Flow-based network monitoring and explain the current state of intrusion detection on Flow-based data.

Chapter 3 explains the security situation and attack possibilities of the so called Web 2.0. We explain how new technologies have made user-friendly and pleasant web applications possible, but on the other hand increased the complexity and opened new attack vectors. Additionally, we give an overview on the variety of attack methods and elaborate on the corresponding defense mechanisms and mitigation techniques. We conclude with some proposals on how the security situation could be improved and pinpoint open research challenges. The vast majority of intrusions, that the systems presented in this thesis try to detect, fall in the category of attacks and vulnerabilities described in this part. The content of this chapter is published in Stritter et al. [22].

In **Chapter 4** we combine multiple Anomaly Detection Algorithms (ADAs) on one machine, to achieve a higher intrusion detection rate than legacy systems which use only one algorithm. The combination of multiple ADAs on a single machine will put a high load on a single system, which has a negative impact on the packet throughput capability. To mitigate this we propose a novel load allocation scheme. With the help of this scheme, packets will skip single algorithm instances, if these instances can not keep up with the incoming packet rate. This way packets can still be analyzed by other ADA instances, and uncontrolled packet drops are mitigated. The content of this chapter is published in Berger et al. [23].

In **Chapter 5** we propose two approaches for preprocessing HTTP data before analysis for network monitoring appliances in general and NIDS in particular. IPFIX Flows have become the de facto standard for exporting Flow-based data in network monitoring (cf. Section 2.2). Usually, IPFIX Flows only contain packet header based data. Thus, our first approach extends the IPFIX protocol and includes, as an example for application-layer data, HTTP elements into IPFIX Flows as own Information Element (IE) fields. The exported IPFIX Flows can then be used by network monitoring systems and, because of the significantly reduced amount of exported data, promise a faster and more efficient analysis. We proposed the developed IPFIX IEs for standardization to the Internet Assigned Numbers Authority (IANA). By now they are part of the official set of IEs of the IPFIX protocol. In our second approach we reduce incoming HTTP data for subsequent packet-based NIDS. The goal hereby

is to reduce the amount of traffic data to be analyzed, but retain all data relevant for intrusion detection. Following the principle of heavy-tailed internet traffic, we show that it is enough to use the first *N* bytes of an HTTP connection to detect most of the intrusions. The content of this chapter has been published in Erlacher et al. [24] and Erlacher and Dressler [25].

In **Chapter 6** we propose a novel NIDS called FIXIDS. FIXIDS takes advantage of the HTTP-related IPFIX IE fields introduced in Chapter 5 and performs signature-based intrusion detection on such IPFIX Flows. FIXIDS uses HTTP-related signatures from the widely used NIDS Snort and applies the contained signature patterns to the corresponding HTTP IE fields. Because of the considerably lower amount of data that has to be analyzed, FIXIDS proves to be significantly faster in terms of network throughput compared to traditional NIDS, while retaining the same detection accuracy and precision as Snort. We additionally show, how FIXIDS can be used to remove a substantial part of the load of a legacy NIDS, which is already deployed. The content of this chapter is published in Erlacher and Dressler [26] and Erlacher and Dressler [27].

In **Chapter 7** we present the automatic attack traffic generator GENESIDS. Evaluating signature-based NIDS is a difficult task, especially attack coverage evaluation experiments. The problem with most publicly available network traces is their old age and, if they contain any application-layer payload at all, the low number of contained attacks. Thus, such traces are not helpful to prove that a newly developed NIDS prototype has a comprehensive detection coverage. GENESIDS uses Snort signatures as attack descriptions and generates traffic that contains the patterns included in these signatures. Therefore, it allows not only to generate traffic from thousands of signatures from the database of Snort rules, but also supports the generation of application-scenario-specific traffic, by writing own attack descriptions using the Snort syntax. The content of this chapter is published in Erlacher and Dressler [28] and Erlacher and Dressler [29].

1.4 Publications

In the following I give a short presentation of all of the scientific work published during my PhD studies, stating my personal contributions to the publications relevant for this thesis. The publications are categorized into two subsections: Section 1.4.1 contains the publications that this thesis is based on, and Section 1.4.2 contains publications I have contributed to during my PhD studies, but which are not related to this thesis.

1.4.1 Publications This Thesis Is Based On

- B. Stritter, F. Freiling, H. König, R. Rietz, S. Ullrich, A. von Gernler, F. Erlacher, and F. Dressler, "Cleaning up Web 2.0's Security Mess - at Least Partly," *IEEE Security & Privacy*, vol. 14, no. 2, pp. 48–57, Mar. 2016

This magazine article was published as fruit of discussions during the Federal Ministry for Education and Research (Bundesministerium für Bildung und Forschung, BMBF) sponsored project "Padiofire".[7] While I was only marginally involved in writing the article itself, I was always part of the meetings and discussions.

- M. Berger, F. Erlacher, C. Sommer, and F. Dressler, "Adaptive Load Allocation for Combining Anomaly Detectors Using Controlled Skips," in *3rd IEEE International Conference on Computing, Networking and Communications (ICNC 2014), CNC Workshop*, Honolulu, HI: IEEE, Feb. 2014, pp. 792–796

My contributions to this conference paper are the writing of the paper itself, the fine tuning of the load allocation scheme, the implementation of one of the ADAs and the major part of the evaluation as presented in the paper.

- F. Erlacher, W. Estgfaeller, and F. Dressler, "Improving Network Monitoring Through Aggregation of HTTP/1.1 Dialogs in IPFIX," in *41st IEEE Conference on Local Computer Networks (LCN 2016)*, Dubai, UAE: IEEE, Nov. 2016, pp. 543–546

My contributions to this conference paper are the supervising of the Masters Thesis, which was the main input for this publication, designing and execution of parts of the evaluation experiments and the writing of the paper itself.

- F. Erlacher and F. Dressler, "High Performance Intrusion Detection Using HTTP-based Payload Aggregation," in *42nd IEEE Conference on Local Computer Networks (LCN 2017)*, Singapore: IEEE, Oct. 2017, pp. 418–425

My contributions to this conference paper are the complete design and implementation of the prototype, planning and execution of all evaluation experiments as well as the writing of the paper itself.

- F. Erlacher and F. Dressler, "How to Test an IDS? GENESIDS: An Automated System for Generating Attack Traffic," in *ACM SIGCOMM 2018, Workshop on Traffic Measurements for Cybersecurity (WTMC 2018)*, Budapest, Hungary: ACM, Aug. 2018, pp. 46–51

[7] www.padiofire.org

My contributions to this conference paper are the complete design and implementation of the prototype, planning and execution of all evaluation experiments as well as the writing of the paper itself.

- F. Erlacher and F. Dressler, "FIXIDS: A High-Speed Signature-based Flow Intrusion Detection System," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2018)*, Taipei, Taiwan: IEEE, Apr. 2018

My contributions to this conference paper are the complete design and implementation of the prototype, planning and execution of all evaluation experiments as well as the writing of the paper itself.

- F. Erlacher and F. Dressler, "On High-Speed Flow-based Intrusion Detection using Snort-compatible Signatures," *IEEE Transactions on Dependable and Secure Computing*, submitted

My contributions to this conference paper are the complete design and implementation of the prototype, planning and execution of all evaluation experiments as well as the writing of the paper itself.

In the following a short list of smaller contributions to regional workshops and demo papers I published. The aim of these publications was to foster discussions about the topic and get further input about the presented subject.

- F. Erlacher, "Network Monitoring for Todays Internet," in *International Conference on Networked Systems (NetSys 2015), PhD Forum*, Cottbus, Germany, Mar. 2015
- F. Erlacher, S. Woertz, and F. Dressler, "A TLS Interception Proxy with Real-Time Libpcap Export," in *41st IEEE Conference on Local Computer Networks (LCN 2016), Demo Session*, Dubai, UAE: IEEE, Nov. 2016
- F. Erlacher and F. Dressler, "Testing IDS using GENESIDS: Realistic Mixed Traffic Generation for IDS Evaluation," in *ACM SIGCOMM 2018, Demo Session*, Budapest, Hungary: ACM, Aug. 2018, pp. 153–155

1.4.2 Publications Not Related to This Thesis

- M. Segata, B. Bloessl, S. Joerer, F. Erlacher, M. Mutschlechner, F. Klingler, C. Sommer, R. Lo Cigno, and F. Dressler, "Shadowing or Multi-Path Fading: Which Dominates in Inter-Vehicle Communication?" University of Innsbruck, Institute of Computer Science, Technical Report CCS-2013-03, Jun. 2013

- F. Erlacher, F. Klingler, C. Sommer, and F. Dressler, "On the Impact of Street Width on 5.9 GHz Radio Signal Propagation in Vehicular Networks," in *11th IEEE/IFIP Conference on Wireless On demand Network Systems and Services (WONS 2014)*, Obergurgl, Austria: IEEE, Apr. 2014, pp. 143–146

- M. Mutschlechner, F. Klingler, F. Erlacher, F. Hagenauer, M. Kiessling, and F. Dressler, "Reliable Communication using Erasure Codes for Monitoring Bats in the Wild," in *33rd IEEE Conference on Computer Communications (INFOCOM 2014), Student Activities*, Toronto, Canada: IEEE, Apr. 2014, pp. 189–190

- F. Erlacher, B. Weber, J.-T. Fischer, and F. Dressler, "AvaRange - Using Sensor Network Ranging Techniques to Explore the Dynamics of Avalanches," in *12th IEEE/IFIP Conference on Wireless On demand Network Systems and Services (WONS 2016)*, Cortina d'Ampezzo, Italy: IEEE, Jan. 2016, pp. 120–123

- F. Erlacher, F. Dressler, and J.-T. Fischer, "New Insights on a Sensor Network based Measurement Platform for Avalanche Dynamics," in *International Snow Science Workshop (ISSW 2018)*, Innsbruck, Austria, Oct. 2018, pp. 31–34

1.5 A Note on Moral Implications of Network Monitoring

NIDS are part of a bigger set of network monitoring tools [37]. The monitored traffic will, in the majority of the application scenarios for network monitoring tools, contain privacy sensitive data. This means that such tools can be (mis)used for personal surveillance and abuse of private data. Thereby, the intended purpose of the network monitoring tool is of no importance. What matters is how this tool is finally applied and how the gathered data is processed and used.

The United Nations (UN) Universal Declaration of Human Rights (UDHR) [38] under Article 12 states that "No one shall be subjected to arbitrary interference with his privacy, ... or correspondence .... Every one has the right to the protection of the law against such interference or attacks." The UDHR is not legally binding, thus, it depends on the local jurisdiction if this basic human right is enforced at all and to what extend.

Despite the publication of questionable electronic surveillance methods and tools applied by governmental intelligence agencies around the world [39], the legislation concerning mass surveillance and privacy is currently progressing in a direction of favoring mass surveillance whilst narrowing privacy [40].

We have not yet reached scenarios as described in the well-known dystopic novel "Nineteen Eighty-Four" by George Orwell [41]. But powerful states like China already

successfully operate massive firewalls censoring the internet [42]. And concrete plans of the Chinese government to introduce a social credit system [43], which regulates access to financial and public services based on the everyday behavior and discipline of a person, bring reality very close to orwellian fantasies.

If network monitoring in general or the methods proposed in this thesis help or harm digital privacy depends mostly on the user. This entails that individuals involved in research and application of network monitoring tools, should be aware of the implications that their decisions and actions have. Almost all computer science societies and associations have codes of conduct and guidelines.[8] [9] [10] Most do mention privacy in relation to experiments (e.g., protection of privacy of participants of experiments when publishing results). But none explicitly elaborates on the responsibility of individuals in terms of the moral implications that their work might have.

In research fields with a more direct impact on the well-being of natural persons (e.g., biomedical research) the discussion about ethics and moral implications has become everyday business [44]. In computer science and in particular in network monitoring there are only very few research projects focusing on privacy [45], [46], or discussions revolving around the moral implications of the scientific work [47], [48]. It would be desirable if in the future more projects would focus on preserving privacy in network monitoring and if more discussions would revolve around the moral implications of our work.

[8] https://ethics.acm.org/code-of-ethics/

[9] https://www.ieee.org/content/dam/ieee-org/ieee/web/org/about/ieee_code_of_conduct.pdf

[10] https://www.caida.org/publications/papers/2012/menlo_report_actual_formatted/menlo_report_actual_formatted.pdf

Chapter 2

Fundamentals and Related Work

In the first Chapter of this thesis we motivated this work and explained the basics of intrusion detection. In this chapter we will explain the fundamentals this work is built upon and give an overview of current research efforts in this domain. Section 2.1 presents fundamentals and current efforts to improve the network throughput performance of Network Intrusion Detection Systems (NIDS) and corresponding related work. In Section 2.2 we present the general concept of Flow monitoring and Section 2.3 presents the state-of-the-art of how the Flow monitoring concept has been expanded to intrusion detection. Section 2.4 presents Vermont, which is the toolkit that has been used for the development of almost all prototypes developed during this thesis. Finally, Section 2.5 briefly elaborates on how today's monitoring and intrusion detection appliances handle encrypted traffic.

2.1 Improving the Network Throughput Performance of NIDS

There are two basic approaches to increase the network throughput performance of NIDS:

- One is to increase the performance of the analysis process. For anomaly-based intrusion detection this includes mainly improvements on Anomaly Detection Algorithms (ADAs). For signature-based NIDS, the performance of the analysis process is mainly increased by improving the pattern-matching process. Because in this thesis we deal mostly with signature-based systems, we will elaborate on this in Section 2.1.1.
- The second approach to increase the network throughput performance of a NIDS is to reduce the amount of data that it has to analyze. In Section 2.1.2 we present current methods to reduce the traffic data for packet based NIDS.

2.1.1 Improving the Pattern Matching of NIDS

The biggest performance bottleneck during the analysis stage of signature-based NIDS is the pattern-matching process [17], [49], [50]. Patterns in signatures typically consist of strings or Regular Expressions (RegExes) and are compared against the data (or parts of the data) of incoming network traffic. Usually, NIDS apply string pattern matching first, and only if that produces a match, the more expensive regular-expression pattern matching [51] is applied. For string pattern matching [52], as well as regular-expression pattern matching [53], there have been efforts that offload the process to specialized hardware. Nevertheless, the majority of NIDS are deployed using off-the-shelve hardware and, thus, such approaches only have limited usage. Also in this thesis, we focus on off-the-shelve hardware only.

The most commonly used algorithm for string pattern matching in NIDS is the Aho-Corasick algorithm [54]. Although published already in 1975, for improving library bibliographic search programs, it is still in use today. The problem with this algorithm is that it accesses memory frequently and, thus, generates a high number of cache misses, which have a negative impact on the performance. There have been several efforts to improve this: Choi et al. [49] use a small sliding window over the search space and try to discard parts of the text, that will not generate a match, as early as possible. Additionally, they use multiple filters, which allows them to heavily exploit instruction level parallelism. In their evaluation they outperform the original Aho-Corasick algorithm by a factor of 2. Other approaches like Stylianopoulos et al. [50] try to enforce cache locality and exploit modern Single Instruction Multiple Data (SIMD) instructions.

But also the optimization of regular-expression pattern matching has been target of scientific efforts [55]. Most systems match regular-expressions with the help of Deterministic Finite Automatons (DFAs). In theory, this approach is well researched, but practical implementations often suffer from severely high memory consumption. Several studies [56], [57] optimize the automata representation to decrease the memory consumption of such a DFA, leading to a higher processing speed.

Summarizing, there have been several successful efforts to improve the pattern-matching process of NIDS. But the performance improvements reached by the above approaches are not enough to make other performance increasing methods obsolete.

2.1.2 Reducing Network Traffic for Analysis

From all the traffic that a NIDS has to analyze, only a very small part is related to security incidents and will finally trigger an event. For performance reasons, it would be desirable if the traffic could be reduced to this fraction of interest before analysis. But it is generally impossible to pinpoint the exact location of the traffic of interest. Thus, the focus usually lies in reducing the traffic volume for analysis by taking away possibly uninteresting traffic parts.

A relatively easy way, which is applicable at a low performance cost, is to filter traffic by address or port. But firstly, even after this filtering step, the data volume might still be too high for the analysis stage to cope with. And, secondly, network operators usually place NIDS at locations where all bypassing traffic must be analyzed. So additional solutions for reducing the traffic before analysis have to be used.

Another traffic reduction strategy at a relatively low computational cost is packet sampling [58]. According to Zseby et al. [59], packet sampling processes can be divided into two main categories: The first being systematic sampling and the second being random sampling [60]. In systematic sampling, packets are selected according to a deterministic function (e.g., select a packet every every t seconds or every n^{th}

packet in a stream). In random sampling, on the other hand, packets are selected according to a random process (e.g., select every nth packet, where n is a random number).

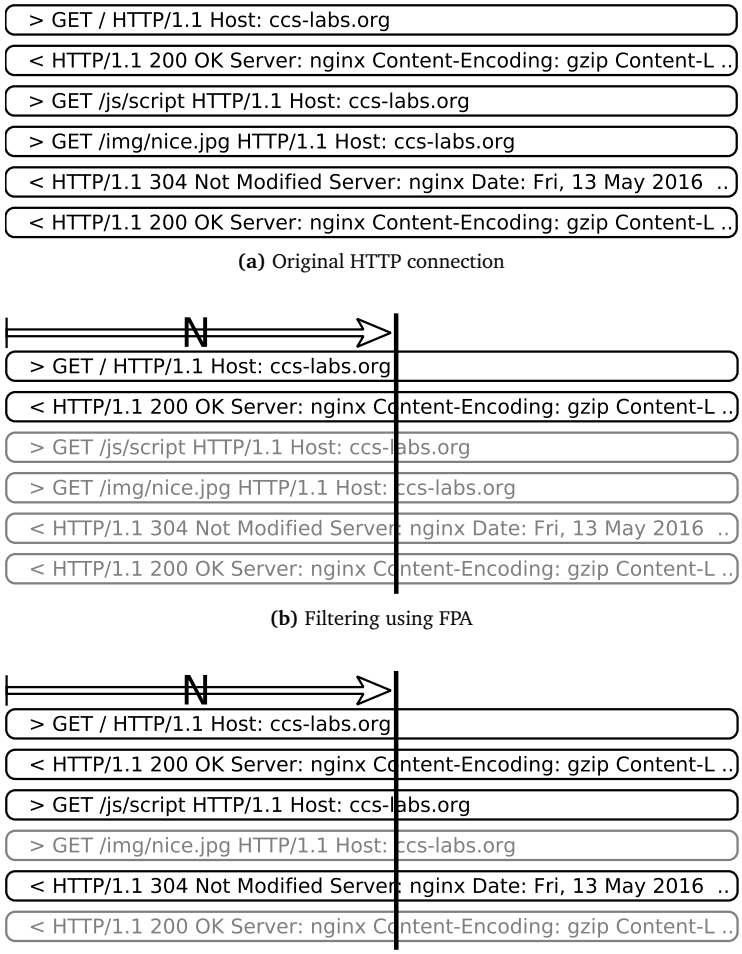
Packet sampling helps reducing the traffic volume but, unfortunately, also significantly reduces the detection quality [61], [62], because generally speaking, data is filtered using mathematical functions instead of caring about its benefit for intrusion detection analysis. This becomes severely evident if the system inspecting the filtered traffic is analyzing the application-layer payload: It might be impossible to decode an application-layer protocol, if not all associated packets are available. If, for example, a missing packet contains an integral part of a Hypertext Transfer Protocol (HTTP) header, the analyzing system will not be able to parse the associated HTTP message and, thus, will miss all intrusions from that specific message, even though possibly only one packet is missing.

To avoid dismissing possibly valuable parts of the traffic, there are several proposals that try to filter traffic by its value for the intrusion detection process. Usually, such approaches try to exploit the heavy-tailed nature of internet traffic [63], [64]. They all have in common that they use the first N bytes of a TCP connection, arguing that the relevant information for intrusion detection is usually located at the beginning of a stream.

This is very similar to early filtering algorithms such as Time Machine [65] or Front Payload Aggregation (FPA) [66] and Dialog-based Payload Aggregation (DPA) [67], where TCP sequence numbers are used to aggregate the first N bytes of TCP connections. While this works well if the application layer contains rather simple protocols like SMTP, it fails for protocols that use one transport protocol session in an interleaved way for control commands and transfer of data. One example is HTTP: There is a semantic correlation between subsequent traffic flows in bidirectional communication (e.g., one (or multiple) requests followed by the corresponding (one or multiple) responses on the same TCP connection). This correlation is lost with these simple approaches.

Figure 2.1 exemplifies this by comparing the retained data of the single filtering methods. Figure 2.1a shows the baseline HTTP connection which contains two pipelined HTTP requests. In all of the illustrations one line represents one HTTP message. FPA retains the first N bytes of the outgoing and incoming direction of the TCP connection (after the handshake). As can be seen in Figure 2.1b, it thus misses 4 HTTP messages.

With DPA, Limmer and Dressler [67] extended the FPA approach to application-layer protocol sessions, by collecting the first N bytes after every TCP direction change. In their evaluation experiments of DPA, they were able to detect 89 % of the intrusion events with only 3.7 % of the traffic (with $N = 2\,\text{kByte}$). As shown in Figure 2.1c, this enables too keep track of multiple HTTP request / response



(a) Original HTTP connection

(b) Filtering using FPA

(c) Filtering using DPA

Figure 2.1 – Filtering of an HTTP connection with different legacy filtering techniques; from [25] ©2017 IEEE.

pairs in one TCP connection, which is the standard behavior with the “keep-live” HTTP feature. Nevertheless, the main weakness of this approach is that it is not able to deal with modern pipelining features in HTTP/1.1 [68] or HTTP/2 [69] (e.g., Dynamic Adaptive Streaming over HTTP (DASH) [70] or HTTP Live Streaming (HLS) [71]). Thus, in our example, DPA misses the pipelined HTTP request message and the corresponding response.

2.2 Flow Monitoring

A common approach to enable loss-free traffic analysis in high-speed networks is Flow-based monitoring [72]. With this concept, statistics of a traffic flow are

aggregated and exported to a network monitoring appliance. In Brownlee et al. [73] the concept of a Flow is described "like an artificial logical equivalent to a call or connection". The Internet Protocol Flow Information Export (IPFIX) Working Group (WG) [74] applies this definition to IP Networks in the following way (an Observation Point is "a location in the network where IP packets can be observed"):

> A Flow is defined as a set of IP packets passing an Observation Point in the network during a certain time interval. All packets belonging to a particular Flow have a set of common properties. Each property is defined as the result of applying a function to the values of:
>
> 1. one or more packet header fields (e.g., destination IP address), transport header fields (e.g., destination port number), or application header fields [...].
>
> 2. one or more characteristics of the packet itself (e.g., number of MPLS labels, etc...).
>
> 3. one or more of fields derived from packet treatment (e.g., next hop IP address, the output interface, etc...).
>
> A packet is defined as belonging to a Flow if it completely satisfies all the defined properties of the Flow.

Depending on the application scenario, different common properties to select the packets can be chosen. These properties are also called Flow keys. Often used Flow keys are IP source and destination address in combination with TCP source and destination port. The exported Flow is composed of different Flow fields. This set of fields contains the Flow keys and, depending on the configuration, also other properties, statistics and information about the aggregated packets. Examples of other fields are the Flow duration or the number of monitored packets for this Flow.

Figure 2.2 shows a typical Flow monitoring setup. The network operator will place the Observation Point at a place in its network, where the packets of interest pass by (e.g., a mirroring port at a switch). Flow Records (or simply Flows[11]) are then generated from the packets by the Flow Metering process (this process is also called Flow aggregation) and exported by the Flow Export process. During the

[11] Strictly speaking there is a difference between Flow Records and Flows, as one Flow, containing a lot of information, might be distributed over two or multiple Flow records.

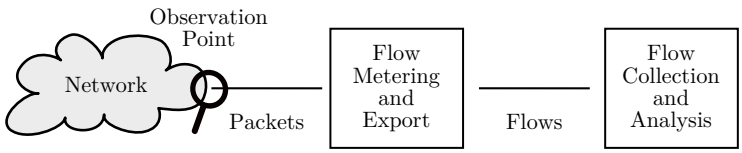


Figure 2.2 – Typical Flow monitoring scenario.

metering process the information of bypassing packets is stored in buffers, one buffer for a set of matching Flow keys. The final Flows for every buffer are exported if the Flow expires. Expiration takes place if one of the following events occur:

- **Passive timeout expiration:** A Flow reaches its passive timeout limit, if the Flow metering process does not receive any new packets, belonging to a certain Flow, for a configurable amount of time.
- **Active timeout expiration:** A Flow reaches its active timeout limit, if the time from the first packet to the current point in time exceeds the configured active timeout.
- **Flow end:** If the transport protocol session is terminated. e.g., for TCP if a packet with FIN or RST flag is observed.

While Flow Metering and Flow Export are two different processes, they are usually done by the same appliance, typically a switch equipped with the corresponding capabilities. Such a combined appliance is also called a Flow probe. The appliance receiving the exported Flow Records is called a Flow Collector. The Flow Collector typically stores these Flows in a database for later analysis or analyzes them directly. In Figure 2.2 the Flow analysis is carried out in the same location as the Flow collection. This, of course, is not mandatory as the analysis can be carried out also elsewhere (using e.g., stored Flows from a database).

When analyzing exported Flow data, it is important to consider that the Flow Records may contain measurement artifacts as described in Koegel [75] and Hofstede et al. [76], [77]. These artifacts reach from imprecise timing information (e.g., in Flow duration field), to missing TCP flags, to wrong information in byte counters (because e.g., padding bytes are not stripped). Because the artifacts differ depending on the Flow Metering implementation, network operators should pay attention which fields are of importance for the Flow analysis and if they are reported correctly by the used appliance.

In the next two sections we present the two most prominent protocols for Flow data export: The NetFlow protocol is briefly presented in Section 2.2.1 and, because of major interest for this thesis in a more comprehensive way, the Internet Protocol Flow Information Export (IPFIX) protocol in Section 2.2.2.

2.2.1 Cisco NetFlow

NetFlow is a proprietary protocol for export of Flow information by Cisco. In its early versions NetFlow had only a fixed set of fields it could export, with NetFlow v5 being its most popular representative. With NetFlow v9 [78] Cisco introduced a much more flexible Flow export protocol. The fields, included in a Flow Record, can now

be adapted to the application scenario. This was made possible by the introduction of Flow Templates. Flow Templates provide a description of the structure of a Flow Record and are exported by the Flow Exporter along with the Flow Records. This allows a Flow Collector to decode Flow Records of different field compositions. There are many networking appliances, not only manufactured by Cisco, which support the export of Flow Records via the NetFlow protocol. Pleas note, that what is sometimes denoted as NetFlow v10 has actually nothing to do with NetFlow, but is used to identify the IPFIX protocol.

2.2.2 Internet Protocol Flow Information Export (IPFIX)

In 2001 the Internet Engineering Task Force (IETF) chartered the IPFIX WG [79], [80]. After an initial surveying and requirements finding phase, the IPFIX WG decided to use NetFlow v9 as a starting point, and they developed the IPFIX protocol, which was finally published in 2008 [74]. Same as NetFlow v9, also the IPFIX protocol uses templates to enable flexible Flow Record definitions. The single fields of an IPFIX Flow Record are called Information Elements (IEs). The initial IPFIX Request for Comments (RFC) [81] already lists 238 different IEs. The Internet Assigned Numbers Authority (IANA) maintains a registry of approved and standardized IPFIX IEs, which, as of February 2019, contains almost 500 well defined entries. Traditionally, these fields can be deducted from packet headers. As explained in Chapter 5, application-layer-based IEs have only been introduced recently.

But an IPFIX Flow Record can not only consist of IANA registered IEs. The IPFIX standard allows also for so called enterprise specific IEs, which allow to define a Flow Record field, which was not yet described, but possibly necessary for an application specific task. Another important distinction between IPFIX and NetFlow is that IPFIX IEs fields must not have a fixed length. When needed, IEs can be defined to have a variable length. This makes the definition of an IE, where the length is unknown in advance (e.g., a Uniform Resource Identifier (URI)), in a Flow Record much simpler.

The following is a textual representation of the contents of a possible IPFIX Template Record:

```
+--- Ipfix Template Record (template id=999, # of fields=6)
'- sourceIPv4Address            (id=8, length=4)
'- destinationIPv4Address       (id=12, length=4)
'- sourceTransportPort          (id=7, length=2)
'- destinationTransportPort     (id=11, length=2)
'- packetTotalCount             (id=86, length=8)
'- octetDeltaCount              (id=1, length=8)
+---
```

With the help of a Template Record, an IPFIX Flow Metering process can tell an IPFIX Flow Collector how to interpret single Flow Records. A Template contains an exact description of every single field contained in the Flow Record and its field length. For most IANA standardized fields, the unique IE ID would suffice to describe a field. But fields may also be smaller than the defined length, thus, it is important that the Flow Template also contains the field length. The above IPFIX Template only contains descriptions for IANA standardized fields. A Flow Record using the format described with the above IPFIX Template could look like the following textual description:

```
+--- Ipfix Data Record (template id=999)
'- sourceIPv4Address            :10.0.0.15
'- destinationIPv4Address       :192.168.0.3
'- sourceTransportPort          :50488
'- destinationTransportPort     :80
'- packetTotalCount             :13
'- octetDeltaCount              :1895
+---
```

Firstly it states the corresponding template id. This way the Flow Collector knows in which template to lookup the exact structure of this IPFIX Flow Record. Then follow the single IPFIX IEs: The first four are the Flow keys, namely IP source and destination address, TCP source and destination port. Then follows a field denoting how many packets have been captured and used to aggregate the information contained in this Flow. Finally, the `octetDeltaCount` field denotes the number of bytes of the packets belonging to this Flow (including IP header and payload).

Throughout this thesis we will capitalize the word Flow (as in IPFIX Flow) to emphasize the distinction from other meanings of the word flow (as in TCP flow).

2.3 Flow-Based Intrusion Detection

Network Flows are particularly attractive for intrusion detection because the amount of data to analyze is only a fraction of the full network data (in the magnitude of 0.1% [82]), especially when considering the steadily increasing network speeds. However, the lower amount of data, notably the almost complete lack of application-layer data, make it hard to reach the same detection accuracy as Deep Packet Inspection (DPI)-based intrusion detection analysis on full network traffic [83]. But there are some attack classes which are particularly suited for Flow-based intrusion detection, namely attack classes that can be detected by looking at packet header and / or traffic statistics. According to Sperotto et al. [84] the following four classes of attacks can be detected with Flow-based intrusion detection approaches:

1. Denial of Service (DOS) attacks
2. Network scans
3. Worms
4. Botnets

The first and most prominent class of network attacks which can be successfully detected with Flow-based approaches are brute-force DOS attacks. The target of such attacks is to make the attacked system unavailable by overwhelming it with a flood of requests that should overload the resources of the system. Although DOS attacks are known since decades, they are still very effective and, because of so called booter services, which offer Distributed Denial of Service (DDOS) attacks for hire [5], [85], evermore easy to realize. According to Sadre et al. [86], DOS attacks can be detected with a Flow-based intrusion detection system by monitoring the number of packets per Flow, the number of bytes per Flow and the Flow duration. The reason for this is that such attacks will generate a high number of short connections and, thus, the number of packets per Flow and the number of bytes per Flow will decrease significantly, while the number of exported Flows with a short duration will rise considerably.

Hofstede et al. [87] used the same metrics to construct a model for their network of interest when no attacks occur. Their prototype then detects DOS attacks by looking for deviations in the current metrics from this model. The reaction of their intrusion detection model to such attacks is to directly blacklist and block the source addresses of the attacking hosts at the attached firewall. Because their system is located at the Flow Metering appliance, they can also prevent flooding of the Flow Collector by filtering Flows that are part of the attack and, thus, not of interest for the Flow Collector.

The second class of attacks are network scans. Using network scans, an attacker can find out which services a system is running and evaluate if the system is vulnerable to certain exploits. This means that a scan itself does not cause any harm, but it is a strong warning sign that an attack might be imminent. Network scans are categorized into horizontal and vertical scans. Horizontal scans denote the scanning of one service or port on multiple systems. Vertical scans, on the other hand, typically scan a single system for multiple services or ports. Either way, when looking at it from a Flow monitoring perspective, network scans create a large number of Flows with a small number of source addresses. Wagner and Plattner [88] and Nychis et al. [89] take advantage of this by calculating an entropy over the source addresses of monitored Flows. In the case of network scans, this entropy will decrease drastically and can thus be used for detection of such scans. To hinder easy detection, attackers often use slow scans, thus, avoiding fast increases in source address observations and

therefore making detection methods as described above useless. This has recently been tackled by Ring et al. [90]. They take into consideration the knowledge of the observed network in terms of offered services and open ports. This means that unidirectional Flows, without a response from an external host, directed to non-existing addresses on the internal network or to closed ports on the internal network are very likely port scans. If such attempts from a single source host accumulate over time, this host is very likely attempting network scans.

The third class of Flow-based detectable attacks are worms. Worms are computer programs that replicate themselves to spread to as much computer systems as possible. They usually use unpatched vulnerabilities to gain access and nest in computer systems. Even without doing any intentional harm to the infected computers, worms can cause major disruption by overloading the network because of the traffic that the very fast spreading itself causes [91]. From a detection perspective, worms are very similar to horizontal scans. In fact, to find a suitable target to infect, worms randomly scan hosts to find vulnerable victims. There have been different approaches to detect worms on a Flow basis. In Dressler et al. [92] honeypots are used to generate fingerprints of worm attacks using a combination of the destination port number, timing information, number of used connections and number of transmitted bytes. Such fingerprints can then easily be compared to Flow metrics and, as their evaluation shows, reliably detect such worms. A different approach was chosen by Abdulla et al. [93]. They exploit the fact that most of the traffic caused by worms is initiated without a Domain Name System (DNS) request. Thus, they observe DNS related metrics in Flows and if, for a host, the number of such metrics decreases significantly, a worm alarm is triggered.

The fourth class are botnets. Botnets are composed of a large number of geographically and administratively distributed computers, which are infected by a software that allows remote control by a so called *bot master*. The bot master can use botnets for various tasks. Examples include coordinated attacks like DDOS campaigns [85] or the mining of cryptocurrencies [94]. Usually, botnets are detected by looking for so called *Command and Control Communication (CCC)* between the bot master and the single bots of the botnet. Many botnets used and still use Internet Relay Chat (IRC) for their CCC. This motivated Karasaridis et al. [95] to craft a dynamic Flow-based model, characterizing CCC based on metrics like remote IP addresses and ports, as well as number of Flows, packets and bytes. Once such models are established, they are able to detect such communication successfully in Internet Service Provider (ISP) sized networks. A more recent approach [96] makes the detection independent of IRC, by looking for typical periodic messages from the bot master to the single bots during the attack preparation and coordination phase and the subsequent periodical beacon messages from the single bots to the bot

master during the attack. These traffic patterns differ significantly from the typical server to client web traffic and can thus be successfully detected.

The above examples show that Flow-based intrusion detection offers fast and scalable methods for detection of a broad variety of attacks. These methods can also be used for a more comprehensive intrusion detection method: SSHCure [97]–[99] is a system, solely relying on Flows, that not only is able to detect single attack phases of Secure Shell (SSH) brute-force attacks but, most importantly, is able to tell if an attack was successful and the targeted machine was compromised.

As the survey on Flow-based intrusion detection by Sperotto et al. [84] points out "the complete absence of payload should still be perceived as the main drawback of Flow-based approaches". This is one of the challenges that this thesis resolves, with the final result presented in Chapter 6 with FIXIDS which performs signature-based intrusion detection on IPFIX Flows containing HTTP-related IEs.

Nevertheless, it is important to note that Flow-based intrusion detection can not achieve the overall precision and accuracy of packet-based intrusion detection and, thus, can not replace it. Flow-based intrusion detection should be a complement in high-throughput scenarios, where network traffic is monitored and exported in form of Flows Records and packet-based intrusion detection would not scale.

2.4 The Vermont Network Monitoring Toolkit

The VERsatile MONitoring Toolkit (Vermont) is a modular framework for a variety of network monitoring tasks. It was initially development during the *History* project [100] as an IPFIX standard-compliant, open-source Flow probe and Flow Exporter,

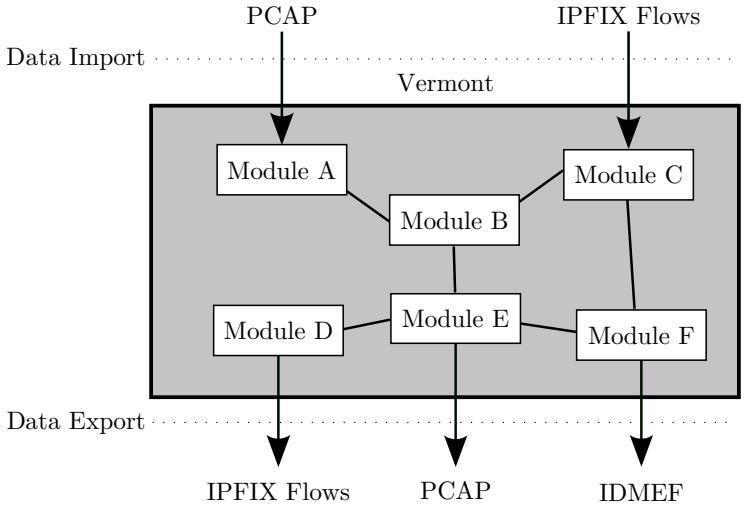


Figure 2.3 – The VERsatile MONitoring Toolkit (Vermont).

running on off-the-shelve hardware. According to Lampert et al. [101] the “design principles were:

- IPFIX/PSAMP compliant monitoring and data export
- Rule-based flow metering and aggregation
- Multiprocessor support
- High monitoring performance

”

As depicted in Figure 2.3, the multiprocessor support and, thus, also the high monitoring performance, is achieved by dividing the functionality among different modules. Every single module runs as an own thread on a multicore processor. Modules can be combined arbitrarily. The only constraint is that the input and output format of connected modules must match. This approach enables the reusability of single module functionalities and easy implementation of additional features. The variety of modules also allows for a multitude of input and output formats, ranging from real-time packet capturing on live networks to reading from stored pcap traces to receiving IPFIX Flows. Meanwhile, the functionality of Vermont has extended far beyond what is listed above. This includes an anomaly-based NIDS module (cf. Chapter 4), a signature-based Flow-based NIDS module (cf. Chapter 6) and an IPFIX Flow filtering module, to just name a few.

For this thesis, the most relevant functionalities are the IPFIX Flow probe and IPFIX Flow Exporter functionality of Vermont. An example configuration is depicted in Figure 2.4. In this configuration, Vermont reads packets from a Network Interface Controller (NIC) with the help of the *observer* module. The output format of the observer module is packets, which matches with the input format of the next module, namely the *packetQueue*. This module acts as a FIFO queue of configurable size and should mitigate differences in the rate at which the single modules can process the packets. The packetQueue forwards the packets to the *packetAggregator* module, which aggregates them to IPFIX Flows according to the configuration. The IPFIX

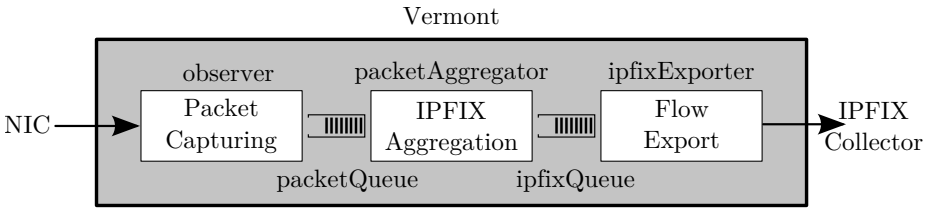


Figure 2.4 – Example configuration of Vermont acting as an IPFIX Flow probe and Exporter.

Flows are then handed to the *ipfixQueue*, which buffers and forwards them to the *ipfixExporter*. This module sends the IPFIX Flows to an external IPFIX Collector.

All the configuration details are defined in the configuration file. Vermont uses the Extensible Markup Language (XML) format for this file. An example configuration file corresponding to Figure 2.4 looks like the following:

```
<ipfixConfig xmlns="urn:ietf:params:xml:ns:ipfix-config">
        <sensorManager id="99">
                <checkinterval>1</checkinterval>
                <outputfile>sensor_output.xml</outputfile>
        </sensorManager>
        <observer id="1">
                <interface>eth0</interface>
                <captureLength>1500</captureLength>
                <next>2</next>
        </observer>
        <packetQueue id="2">
                <maxSize>5000</maxSize>
                <next>3</next>
        </packetQueue>
        <packetAggregator id="3">
                <rule>
                        <templateId>999</templateId>
                        <flowKey>
                                <ieName>sourceIPv4Address</ieName>
                        </flowKey>
                        <flowKey>
                                <ieName>destinationIPv4Address</ieName>
                        </flowKey>
                        <flowKey>
                                <ieName>protocolIdentifier</ieName>
                        </flowKey>
                        <flowKey>
                                <ieName>sourceTransportPort</ieName>
                        </flowKey>
                        <flowKey>
                                <ieName>destinationTransportPort</ieName>
                        </flowKey>
                        <nonFlowKey>
                                <ieName>flowStartNanoSeconds</ieName>
                        </nonFlowKey>
                        <nonFlowKey>
                                <ieName>flowEndNanoSeconds</ieName>
                        </nonFlowKey>
```

```
                </rule>
                <expiration>
                        <inactiveTimeout unit="sec">60</inactiveTimeout>
                        <activeTimeout unit="sec">100</activeTimeout>
                </expiration>
                <pollInterval unit="msec">100</pollInterval>
                <next>4</next>
        </packetAggregator>
        <ipfixQueue id="4">
                <entries>200</entries>
                <next>5</next>
        </ipfixQueue>
        <ipfixExporter id="5">
                <templateRefreshInterval>50</templateRefreshInterval>
                <maxRecordRate>4000</maxRecordRate>
                <sctpReconnectInterval unit="sec">1</sctpReconnectInterval>
                <sctpDataLifetime unit="msec">1000000</sctpDataLifetime>
                <collector>
                        <ipAddress>10.0.0.1</ipAddress>
                        <port>4739</port>
                <transportProtocol>SCTP</transportProtocol>
                </collector>
        </ipfixExporter>
</ipfixConfig>
```

The first module included in this configuration file is the *sensorManager*. This module is responsible for exporting statistics information about all the modules in the configuration. The sensor data included in the exported statistics reaches from Central Processing Unit (CPU) utilization to memory consumption, but also contains module specific information like number of dropped packets for the observer module or number of aggregated IPFIX Flows for the packetAggregator module. In this case, the statistics are exported every second to a file named sensor_output.xml.

Next in the configuration file is the observer module configuration, which includes the configured network interface and the packet capture length. The `next` clause at the end of the single module configurations always denotes the ID of the module the data should be forwarded to. This can possibly also be multiple modules.

The packetQueue module is configured to have a fixed size of 5000 packets. The packetAggregator module configuration contains the rules on how to aggregate packets to IPFIX Flows. In this configuration only one rule is defined. The IE fields contained in the exported Flows are communicated via Flow templates (cf. Section 2.2.2). The template describing this rule is configured to have the template ID 999. The Flow keys for aggregation are the source and destination IP address, the

source and destination TCP port and the transport protocol identifier. These IE fields along with the non Flow key fields (in this case, two fields describing the Flow start time and Flow end time) are all the Fields contained in the exported IPFIX Flows.

Then follows the ipfixQueue module with a fixed size of 200 Flows followed by the ipfixExporter module. The ipfixExporter module contains the configuration how and where to export the IPFIX Flows. It contains definitions how often the IPFIX template records should be transmitted, what the maximum Flow rate is and other transport protocol specific definitions. In this case, the used transport protocol is SCTP.

For more detailed information on Vermont and the modules used in this thesis please refer to the code repository, which also contains a wiki page with additional information: `https://github.com/felixe/ccsVermont/wiki`.

2.5 Intrusion Detection on Encrypted Traffic

Because of the grown awareness for personal privacy and the need for authenticated and confidential communication, most application-layer traffic on the internet is encrypted with Transport Layer Security (TLS) (the latest version being 1.3 [102]). This is not so much a problem for Flow-based intrusion detection, as presented in Section 2.3. With such approaches the information comes from protocol headers which, in most encryption scenarios, remain in cleartext. But the encrypted traffic becomes challenging for novel Flow-based intrusion detection approaches, which include application layer payload, as presented later in this thesis or, more generally, for network operators who still want to monitor their network traffic at the application-layer level.

There have been several efforts to gain valuable insights into encrypted traffic by exploiting statistical properties, which are not changed by the encryption process. They reach from methods to fingerprint websites [103] to traffic classification of encrypted traffic [104]. Most of these approaches have been applied to a relatively small set of websites and corresponding traffic and, thus, only show modest to no success at larger scale [103].

Most relevant NIDS and firewall vendors use so called TLS interception proxies (e.g., Genua's GenuGate[12]) to tackle this challenge. They basically perform a legitimate man-in-the-middle attack by decrypting the incoming application-layer data, exporting it for further analysis and then decrypting the data again and forwarding it to the original recipient. Because no existing, open-source TLS interception proxy was able to export the encrypted payload in the libpcap format, we have built a prototype [31] with the capability of real-time libpcap export. At the time of

[12] `https://www.genua.de/loesungen/high-resistance-firewall-genugate.html`

writing of this thesis, other TLS interception proxies with libpcap export features have been published (e.g., SSLsplit[13]). For the rest of this thesis we assume that application-layer data is provided in cleartext.

[13] https://www.roe.ch/SSLsplit

Chapter 3

Web 2.0 Security

In the previous chapter we learned about fundamentals of NIDS and about related work on how to improve the efficiency of such systems. In this chapter we give an overview of the threats that have been introduced with the Web 2.0 technologies. We also look at current mitigation techniques and point out open research problems.

This chapter is based on the following publication:
B. Stritter, F. Freiling, H. König, R. Rietz, S. Ullrich, A. von Gernler, F. Erlacher, and F. Dressler, "Cleaning up Web 2.0's Security Mess - at Least Partly," *IEEE Security & Privacy*, vol. 14, no. 2, pp. 48–57, Mar. 2016

3.1 Motivation

Before the so called Web 2.0, the World Wide Web (www) consisted of a huge variety of commercial and non-commercial applications: from simple link collections to search engines to web shops to audio and video telephony applications. With the advent of the Web 2.0, internet content became more interactive. Whereas the old web consisted mainly of static pages, the new Web 2.0 is built upon highly interactive web applications, which allow the user to create personal and, thus, more sensitive content.

Multiple scripting languages were invented to facilitate the creation of easy-to-use web applications with JavaScript [105] being the most prominent. Then, browser manufacturers introduced browser plugins like Java and Flash. The focus during the development of these technologies was always on usability; security only played a marginal role. For example, developers tried to limit the interaction of JavaScript with the computing environment outside of the browser, but failed to do the same for browser plugins. Still today, this represents the root cause for a large number of security incidents.

Because of the rapid development and the constantly evolving and extending feature set of browsers, incompatibilities between single features are still omnipresent. Such inconsistencies are also visible when it comes to security: With the complexity of Web 2.0, browser developers realized that there is a need for more security. But security demands have not been met consistently. An example for this is TLS: it does provide data integrity during the transportation phase, but data integrity is not guaranteed anymore as soon as the browser takes over this data.

Traditional security applications to protect against internet threats are (personal or perimeter) firewalls including a NIDS that inspects traffic contents for known vulnerabilities. When security is paramount, system administrators simply turn off features like JavaScript and Flash, providing security at the cost of convenience. However, with the pervasiveness of new interactive technologies, the loss of conve-

nience, by turning off these technologies, has grown to a level where it is no longer acceptable, because Web 2.0 applications are unusable without active content.

The new technological possibilities led also to an increased commercialization of the web, resulting in pervasive user tracking and a broad dissemination of social networks. With the ubiquitousness of the internet, web technologies are also used more and more for control and configuration of devices like industrial systems or the smart home.

The introduction of all these new technologies offers an unprecedented variety of possibilities to create dynamic and interactive web applications. But on the other hand, it completely changed the security landscape, resulting in a complex system, which is hard to thoroughly understand and which led to a multitude of novel attack vectors, especially in the browser. The pervasiveness of the web and the newly achieved ease of use of the internet has also led to an unprecedented dependency on Web 2.0 technologies.

For the rest of this chapter we try to answer the question how to make today's web secure and usable at the same time. We point out the most pressing Web 2.0 security issues, focusing mainly on the browser. We also explore some defense mechanisms on the server and on intermediate appliances. We hope to bring some order into the mess of modern web applications and demonstrate some possible remedies. Finally, in Section 3.4, we point out open research challenges.

3.2 New Attacks of the Web 2.0

With the newly introduced technologies of the Web 2.0 (e.g., Asynchronous JavaScript and XML (AJAX) and Dynamic HTML (DHTML)) browsers have become powerful enough to move many applications from the desktop to the web. The main advantages of this move are easy application maintenance for the manufacturer and no install overhead for the user. The downside is that now business-critical data on one side, and privacy sensitive data on the other side, are shared through the internet. This motivated advertisement and tracking providers to focus on this freely available and huge amount of information to create precise user profiles [106].

The impact of legacy attacks focusing on buffer overflows has decreased due to defense methods like sandboxing [107] and Address Space Layout Randomization (ASLR) [108]. But new features like support for Scalable Vector Graphics (SVG), browser integration of audio and video and the high number of browser plugins has led to an overall rise of web crime [109]. What makes the Web 2.0 particularly appealing for attacks are the increase of users, that have been attracted by the ease of use and functionality due to the new technologies.

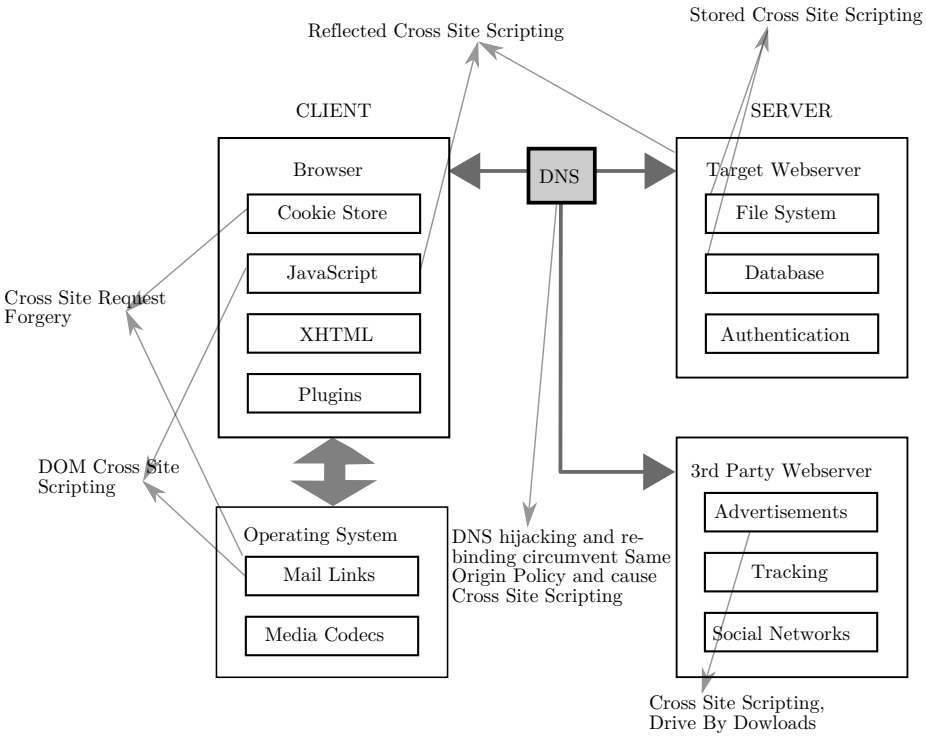


Figure 3.1 – Complex interactions between client and server side aggravate security. Here we describe some attack possibilities in this complex system; derived from [22] ©2016 IEEE.

Figure 3.1 sketches a simplified view of the complex interactions of the modern web architecture. Multiple problems are caused by this, as we will explain in the following.

3.2.1 Merging of Security Domains Inside a Browser

When fetching resources for a web application, the browser contacts web servers from different content providers and, thus, different security domains. These domains reach from the corporate intranet to social networks to advertisement servers. The browser allows interaction between all of them in different ways. Under certain circumstances the browser might even be abused to access the internal network, thus, allowing the attacker to avoid security measures like firewalls. The following are a few examples of commonly used techniques that might lead to successful attacks:

- **Embedding of scripts:** Third-party content like advertisements of social networks are typically included by fetching scripts from external resources without knowing the exact behavior of the scripts. A code snipped like the following could pull in every type of code:

```
<script src:http://acme.com/evil.js>
```

The problem is that these external resources have different levels of trust, but possess the same access privileges on the client. This situation facilitates code execution from unwanted sources. This type of attack is called Cross Site Scripting (XSS) [110].

- **Embedding of iframes:** With the help of iframes, another page can be incorporated into the current page. While scripts inside iframes are more restricted than directly embedded scripts, they are still able to exchange data with other frames by means of *postMessage* Application Programming Interface (API). This can be used for user interface redressing attacks like the following: The web page fetches a script from an advertisement company, which, instead of an advertisement, shows a fake login dialog to steal login credentials. Another way to use iframes for malicious intents is to visually overlaying different parts of a web page, making the user click unintendedly on malicious content. This is also called *Clickjacking* [111].

- **Shared cookie storage:** While HTTP was designed as a stateless protocol, almost all web applications preserve states of sessions, usually with the help of cookies. Cookies are very small files containing key-value pairs with information about the session. They are stored in the browser and sent back with each server request. The problem is that cookies are also sent back if such a request comes from a third-party site. This enables attackers to inject data into existing sessions. Such an attack is called Cross Site Request Forgery (CSRF) [112].

3.2.2 Incomplete or Conflicting Standards

Web standards like HTTP or Hypertext Markup Language (HTML) often fail to specify how to handle ambiguous or erroneous content. This makes the interpretation browser dependent.

It is, for example, possible to specify the character set of the HTML content in various places: in the HTTP header, using a Byte Order Mark (BOM) or via various tags inside the HTML content. The following example HTTP response shows how this could look like:

```
HTTP/1.0 200 OK
Content-type: text/html; charset=utf8

+/v8
+ADw-script+AD4-
```

```
alert(1)+ADs-
+ADw-/script+AD4-
```

In the HTTP header, the charset is defined as being UTF-8. However, the HTTP body starts with the UTF-7 BOM. If the '+ADw-', '+AD4-' and '+ADS-' strings are interpreted as '<','>' and ';' this HTTP body content results in JavaScript code, while, otherwise this looks simply like scrambled text. Because of the different interpretations of such character set specifications, this is often used by attackers to mislead NIDS [113].

3.2.3 Unjustified Trust in the DNS and Public Key Infrastructures

One of the vital elements of browser security is the Same Origin Policy (SOP). The idea is that if different contents, like cookies or frames, belong to the same origin, they are allowed to interact freely among each other. Otherwise, interaction between contents is severely restricted. The problem is that the origin is derived from the host or domain name using the Domain Name System (DNS). Unfortunately, DNS, in its original form, is not guaranteed to be authoritative and correct because queries can be hijacked and answered by possibly anyone.

DNSSec [114] is designed as a remedy to this. Hereby, DNS replies are signed and, thus, can be verified by the receiver for correctness. But there are a couple of problems with DNSSec: Firstly, as DNSSec is far from being supported everywhere, queries can still be hijacked. Secondly, trust in DNS servers might be misplaced, as a rouge DNS server can claim any IP address to be the address for a host name it manages. Third, at the core of DNSSec is the trust in Public Key Infrastructures (PKIs), which might also be misplaced as we will explain in the next paragraph.

PKIs are not only the basis for DNSSec but, more importantly also for TLS and, thus, the encrypted version of HTTP namely HTTPS. While the fragility of PKIs has always been known, the lack of more secure alternatives led to blindly trust them. However, in 2011 two compromises of Certification Authoritys (CAs) and the abuse of certificates issued by mistake for intermediate CAs revealed this fragility to the whole world [115]. The mistakenly issued certificates enabled attackers to issue unauthorized but valid certificates for high-profile sites like google.com.

Attacks on DNS and PKIs are not specific to the Web 2.0, but the reliance of the modern web these technologies makes such attacks much more attractive today than in the past.

3.3 Practical Mitigation Methods Today

While the attempt to secure an environment as complex and messy as the Web 2.0 seems pointless, there are some promising solutions. We classify these approaches

according to the deployment place in the network structure: at the browser, at the server or in intermediate systems.

3.3.1 Browser-Side Approaches

Protection against known malicious sites is usually carried out by using Uniform Resource Locator (URL) blacklists like Google Safebrowsing[14], which are directly implemented in the browser and periodically updated. Nevertheless, there will always be sites not included in such a database. In that case a locally installed virus scanner should be able to detect downloaded malware. Modern virus scanners with a broader functionality (nowadays called security suites) usually also apply heuristics to detect "abnormal" system behavior.

A different approach is to separate the browser from the rest of the system by means of dedicated virtual machines or sandboxing. While this prevents malware from damaging the local system, it is not a remedy against attacks, which include the internet only: *XSS*, *CSRF* or *Clickjacking* attacks manipulate data on the internet to steal access credentials, for example. A very successful mitigation technique to prevent such attacks is the renunciation on scripts. The browser extension *NoScript*[15], for example, restricts the execution of JavaScript and, thus, prevents many of the aforementioned attacks. The problem is that most modern web applications do not work without JavaScript or the like and, thus, the use of NoScript needs exhaustive individual adaption, which most users will refrain to do.

To avoid trusting third-party content like advertisement and tracking, browser plugins like *AdBlockPlus*[16] or *Ghostery*[17] detect and block this content. This has no effect on the usability of the website, but destroys the business model of most websites. Thus, some web portals try to detect the usage of such plugins and, in case of successful detection, refrain from offering any content until the plugin is disabled.

To make *CSRF* useless, the browser plugin *CsFire* [116] promises to remove sensitive information from such requests, while still allowing legitimate cross-site requests like payment applications or single-sign-on solutions. Similarly, *Noxes* [117] analyzes and possibly whitelists the requested web page of cross-site requests. Unfortunately, it has a high false-positive rate and needs frequent fine tuning. A different approach is taken by *JaSPIn* [118]. It tries to profile the JavaScript behavior of individual websites and detect anomalies to this profile. Again, it requires individual tuning for many websites and the profile needs to be recreated whenever a web application is updated.

[14] `https://safebrowsing.google.com/`
[15] `http://noscript.net/`
[16] `http://adblockplus.org/`
[17] `https://www.ghostery.com`

Summarizing, browser side solutions protect against known malware and to some extent against untrusted third-party content and CSRF attacks. Many of the presented solutions offer satisfying protection for experts, but fail for the average user. Here, a comprehensive solution tackling the complexities of the Web 2.0 is needed.

3.3.2 Server-Side Approaches

Ideally, the best protection against most web-based attacks are secure web applications. But this would require the developers to be aware of all the security problems and implement necessary countermeasures. And this fails, mostly due to lack of time and monetary resources.

While thoroughly checking and sanitizing the user input to avoid XSS and SQL injections is effective and should be applied by every web application developer, there are also other effective countermeasures: Alexenko et al. [119] propose to set a server secret for each resource using CSRF tokens. Then, the web application allows only actions that contain this token, making unauthorized CSRF fail.

A different approach has been proposed by the World Wide Web Consortium (W3C) named *Content Security Policy (CSP)*[18]: Every site can suggest to limit the execution possibilities of the browser depending on the needs of the site, e.g., include/exclude scripts, styles or media from certain third-party sites, forbid/allow the execution of dynamic code. If these suggestions are used by the browser to change the execution environment, CSP provides exhaustive protection against XSS. However, until now CSP has not found widespread adoption.

Another, frequently applied approach, are so called Web Application Firewalls (WAFs), which are located in front of the web server. Their task is to check the user input, possibly add and verify CSRF tokens and to detect common attack patterns. While some WAFs have the ability to somewhat automatically adapt themselves to the input data, they still need customization depending on the protected web applications. If configured correctly, a WAF can be an important part in the overall web security concept.

Web security researches have acknowledged that security measures will only be applied if they do not need constant user interaction. Solutions like *S2XS2* [120] or *NonceSpaces* [121] require the operator to separate between trusted application data and untrusted external data. Furthermore, it has to be defined how such untrusted data can be securely included in a dynamically generated web page. Similar to CSP, *BEEP* [122] and *Blueprint* [123] require the operator to establish security policies, which then are enforced in the browser, again, distinguishing between internal and

[18] `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`

external data. The problem is that all of the above solutions require implementation in the web application itself.

Finally, *XSSDS* [124] does not require changes in the web application but, in a learning phase, acquires knowledge about the JavaScript behavior of the specific application and thereafter blocks unknown JavaScript. It also allows filter definition against *reflected XSS*, similar to the filter applied at the browser side. *XSS-GUARD* [125] applies a similar approach. It is a server side proxy that creates the expected page based on benign input and compares the real page to this shadow copy, alerting if anomalies are detected.

3.3.3 Solutions for Intermediate Devices

Firewalls are usually the protection of choice for networks of a certain size. But simply filtering traffic according to packet header information is of no use against web-specific attacks. Only NIDS with DPI [126] functionalities have insights into application-layer protocols like HTTP. The combination of classic firewalls with a NIDS or an Intrusion Prevention System (IPS) is called *Secure Web Gateway (SWG)*, or, in marketing lingo *Next Generation Firewall (NGFW)* or *Unified Threat Management (UTM)*. More advanced systems are also able to inspect the payload of TLS encrypted traffic (see Section 2.5), some can also normalize HTML or remove JavaScript and ActiveX code.

The problem is that such systems fail for high-throughput rates. In scenarios, where the incoming network traffic rate is higher than the rate at which the system is able to analyze the traffic, packets will be tailbacked and finally dropped. The information contained in the dropped packets is unknown to the defense system and, thus, the system will not only miss attacks enclosed in these packets, but also miss attacks in analyzed packets if context information from dropped packets is needed for detection.

Other approaches crawl the web continuously for malware detection, looking for already known patterns but also applying anomaly detection techniques. If malicious websites are found, their address is added to a public blacklist. Such approaches are helpful, but by no means comprehensive, because they can never mimic real-life interactions and, thus, attacks on the application logic like CSRF or Clickjacking can not be detected.

3.3.4 Attack Coverage

Table 3.1 lists the previously presented attacks and summarizes the existing protection solutions according to their placement. Client attacks, which require the attacker to previously get control over server functions or network device infrastructure, are

Table 3.1 – Attack coverage, sorted by placement. ✓= good attack coverage, ∼= partial attack coverage, × = no attack coverage; derived from [22] ©2016 IEEE.

Attack	Browser	Intermediate systems	Server
Major Attacks			
Attacks against servers	×	×	✓
Cross-site scripting	∼	∼	∼
Credential/session prediction	×	×	✓
Session fixation	×	×	✓
Cross-site request forgery	∼	∼	✓
Buffer overflow	✓	∼	✓
Malware	∼	∼	∼
URL redirector abuse	×	×	✓
Minor attacks			
Integer overflows	×	×	×
Content spoofing	×	×	×
Remote file inclusion	×	×	✓
HTTP response splitting	×	✓	✓
HTTP request splitting	×	✓	✓
Null byte injection	×	×	×
Routing detour	×	×	×
XML external entities	×	×	✓

categorized as attacks against servers. The first part of Table 3.1 (Major attacks) is loosely based on the OWASP Top Ten Project.[19]

From this table we can conclude that major attacks can be protected best at the server side. But because, in reality, not all web application providers apply these protection solutions, it is important to implement protection also at the browser side and in intermediate devices.

3.4 Open Research Challenges

While the presented solutions provide some security, novel protection methods are required to provide comprehensive security from today's Web 2.0 threats.

Because there will always be flaws in web applications, it is only reasonable to step up the server-side protection. More effort should be put in protection from application-specific vulnerabilities like insecure cookie flags or protection against CSRF flags.

[19] `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`

There have been some approaches on the distinction between trusted and untrusted content, most prominently CSP, and this helped in protecting against XSS attacks. But all of these approaches require modifications of either the web application itself or adoptions on the browser-side. So there is a need for solutions which require little or no effort in application.

Less work has been put in browser-side solutions. While some, very valuable, metrics are only available at the client-side (application state, runtime information), none of the approaches known to us makes good use of them. This is an area where more research is needed.

Solutions on intermediate devices, like NIDS, are rather limited. This is mainly because only very little information about the web application itself and about the browser is available at this location. They typically detect known malware or block traffic according to blacklists. Particular attention should be given to high-speed networks, because today's complex defense mechanisms can not cope with very high rates and, thus, such NIDS will have to drop packets, which leads to lost data which might contain important context information for the detection of attacks.

Summarizing, the presented approaches do not offer comprehensive protection, require adaption in the browser or in the web application itself, which is very cumbersome even for expert users. In the following, we present areas where we see the need for more research.

3.4.1 Browsers Protection Against Typical Web 2.0 Attacks

Here, more research is needed for solutions which require little or no user configuration. We believe that the solution lies in automatic learning of application behavior and to monitor and enforce this behavior not only in the browser, but also on intermediate systems like firewalls and NIDS. Existing solutions have either static profiles which require continuous manual adaption, or stable profiles which fail to offer comprehensive protection. Research efforts should focus more on taking advantage of application runtime information and other metrics which are available at the browser, but, until now have never really been utilized.

3.4.2 Protection in Intermediate Devices

Also here, research should focus on automated solutions with no manual configurations. Protection solutions must be able to adapt to individual and fast moving patterns of client behavior and changing web applications.

Firewalls on intermediate systems must be capable of protecting multiple web applications without having detailed knowledge about them and without needing extensive manual configuration. This would make configuration intensive WAFs, that can protect only a single web application, obsolete.

Currently, analysis methods on intermediate systems either provide DPI analysis of the application layer only (e.g., signature-based NIDS, cf. Section 1.1), or they focus on the protocol stack up to the application layer, but no system provides both analysis methods in a coupled fashion. Combining these two approaches promises better detection capabilities because of additional analysis possibilities. These capabilities should also include learning and enforcing web application models. Research efforts should take into consideration, that the execution of the web application is by no means deterministic, it strongly depends on the context, e.g., stored cookies or previous browsing history.

3.4.3 Secure and Easy to Use Application Frameworks for the Server-Side

CSP is a start in the right direction, helping to develop policies which restrict the interaction of the web application with external resources. Improvements are needed in coupling such efforts with the application framework, which can provide a lot of additional information regarding external resources and help to foster the policies. Also, the interactions among scripts of different origins are not restricted and CSP can not distinguish between inline script and stored XSS. As a remedy, CSP Level 2 includes nonces which should help distinguish intentionally integrated script elements from XSS. However, the creation of such nonces needs to be implemented with the application frameworks itself.

3.4.4 Rethinking the Interaction Between Browser, Server and Components

While writing this chapter we noticed many inconsistencies in the interaction between the browser and the server. Firstly, browsers fail to communicate to the server detailed context about the requested resources. It would greatly help to apply appropriate security measures if the server and intermediate systems would know e.g., if the requested code would be run as a script or included as an image or style. Until now, only the browser has detailed knowledge about the usage of requested resources. For example, the request for a “script.gif” with a content-type of “image/gif” and a content of “GIF87a=1;alert(1)”, does not contain any information for the server to determine if this resource will be used as an image or executed as a script. An additional example for the need of more information in this request are discrepancies between advertised and detected character set of the resource.

3.5 Lessons Learned

In this chapter we gave a brief insight into the complex world of Web 2.0 security. Increasing the security is not a matter of solving singular problems or coming up with a new solution for a specific attack. To increase the security situation in the Web 2.0, comprehensive solutions are needed, which include all of the involved parties, from browser to intermediate systems to the server.

One possible solution would be to strive for self-adapting firewalls which are able to analyze and understand the Web 2.0 application-layer protocols. But this requires that firewalls and NIDS can efficiently analyze application-layer protocols in today's high-speed networks. This is something that we take care of in the following chapters of this thesis.

In the last part of this chapter we pointed out several more open research problems. One of the most pressing ones is the lack of context information from the browser at the server. Such information would ease the application of appropriate security measures and finally lead to a more secure Web 2.0.

Chapter 4

Combining Anomaly Detectors Using Controlled Skips

In the previous chapter we learned about internet threats that have been introduced by novel Web 2.0 technologies. In this chapter we increase the efficiency of anomaly-based Network Intrusion Detection Systems (NIDS), which are used to mitigate such threats.

This chapter is based on the following publication:
M. Berger, F. Erlacher, C. Sommer, and F. Dressler, "Adaptive Load Allocation for Combining Anomaly Detectors Using Controlled Skips," in *3rd IEEE International Conference on Computing, Networking and Communications (ICNC 2014), CNC Workshop*, Honolulu, HI: IEEE, Feb. 2014, pp. 792–796

4.1 Motivation

As outlined in Chapter 3, modern web technologies require NIDS to be an integral part of the IT security of every network [127]. In this chapter we focus on anomaly-based NIDS. The advantage, compared to signature-based systems, is that they can detect previously unknown attacks. This works because attack traffic usually differs from normal traffic, and this deviation can be identified as an anomaly [127], [128].

Anomaly-based NIDS are based on Anomaly Detection Algorithms (ADAs). There are different types of ADAs with different characteristics [14], [15], [20], [129]–[133]. Each of the ADAs has its own advantages and disadvantages: The algorithm proposed by Mahoney and Chan [129], for example, focuses on packet headers only and, thus, is considerably faster than other ADAs at the cost of a higher false positive rate. The ADA proposed by Mahoney [130], on the other hand, additionally analyzes the application-layer payload, thus, promising a more accurate attack detection capability. But the induced complexity disqualifies this algorithm for high-speed networks.

Because of this, it seems reasonable to combine multiple algorithms to take advantage of all of their benefits. Especially the detection accuracy could be improved by combining the detection results of multiple ADAs. The more so, because a detected anomaly should be considered as a "suggestion" and, thus, the combination of multiple suggestions in a post-processing step would consolidate decisions. This is especially relevant when considering the findings of Section 3.4, where we pointed out the importance of combining multiple security and detection systems, to solve security issues in the complex world of the Web 2.0.

There exist different proposals for systems that take advantage of multiple NIDS: Le et al. [134] combine multiple NIDS spread over multiple machines and focus on how to efficiently distribute the network traffic without losing too much information. Also Sekar et al. [135] focus on solving the issues that arise when distributing NIDS functionalities over different nodes in a network. In Toulouse et al. [136]

a proof of concept is presented of a distributed NIDS which runs analysis at every single intrusion detection node, thus, avoiding a central node which collects and analyzes all detection data. All these approaches have in common that the single NIDS instances are distributed on different machines over the network. Thus, they focus on problems like load balancing over the network and flow distribution without information loss for the single NIDS.

In this chapter we focus on using multiple ADAs on a single machine with multiple cores. To the best of our knowledge, there is no system available that makes use of multiple ADAs on one machine in the context of high-speed network monitoring and intrusion detection. Compared to clustering multiple ADAs over a network, the advantage of our approach is that there is no network flow distribution offset, no loss of information because not all network traffic is available at the single nodes and, finally, the cores of the system are optimally used. The challenge with our approach, is to mitigate the problem of high computational cost that is induced when running multiple ADAs on one machine.

We propose to allocate a dedicated Central Processing Unit (CPU) core for every single ADA and to combine the analysis results of all algorithms. Nevertheless, one problem remains: How to handle the different throughput capabilities of the single ADAs? Before taking a decision on single packets or streams, the system has to wait for the results of every single ADA. This entails that complex algorithms will slow down faster ADAs. Thus, a load allocation scheme is required, which considers the complexity of different algorithms. In particular, fast ADAs should not be slowed down or prevented from receiving data by slower, more complex algorithms. Our approach is to skip slow algorithms if resources are depleted and, thus, making sure that the traffic is still analyzed by faster ADAs in real-time. Our framework features metrics to evaluate the performance of integrated ADAs as well as filtering the traffic by the given anomaly score.

Our system has been implemented as part of the network monitoring toolkit Vermont [101] (see also Section 2.4). We implemented two widely used ADAs and apply a load distribution technique which observes all ADAs and eventually skips single instances in a controlled way.

The contributions of this chapter are the following:

- We use multiple instances of different ADAs to gain more insights about the traffic which leads to a more precise detection accuracy.

- We implemented a load allocation scheme which is able to control multiple, concurrently executed algorithms and eventually skips single instances without slowing down the overall process.

- By using the modular concept of Vermont, we combine anomaly detection with high-speed Flow monitoring providing additional information for efficient post processing.

4.2 Architecture

For the implementation of our proposed system we used the network monitoring toolkit Vermont (cf. Section 2.4). Figure 4.1 shows the architecture of our framework. The packet source is typically a Network Interface Controller (NIC) providing live traffic, but the system allows also to analyze previously saved network traces. Immediately after fetching the packets from the packet source, a list for tag fields is added to every packet. This allows every ADA in the following analysis stage to tag the packets with an anomaly score. Then the packets are handed over to the packet analysis stage.

4.2.1 Packet Analysis

Here, every ADA inspects and tags the bypassing packets. Because most ADAs assign floating point scores to packets, we directly adopt these scores and use them to tag the packets with these values along with a unique instance ID of the ADA. The instance ID is necessary to be able to identify the ADA instance who gave the score in the post-processing step.

Precautions were taken to support different types of algorithms. This includes unsupervised algorithms which do not need any user interaction, but also so called semi-supervised algorithms. This category of ADAs requires an initial training time to build a model of the expected traffic.

For the evaluation of our framework we implemented two ADAs: Firstly we implemented Packet Header Anomaly Detection (PHAD) [129], which is an algorithm that

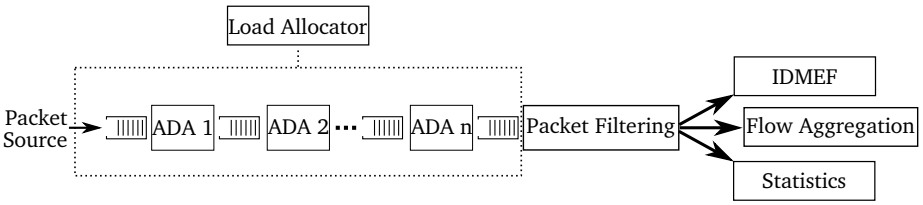


Figure 4.1 – Architecture of the anomaly detection framework. Packets are fed to a pipeline of multiple ADAs, which are controlled by the load allocator. Every ADA, if not overloaded, gives every packet an anomaly score which is later used to filter the packets, which then can be exported in multiple ways; derived from [23] ©2014 IEEE.

only examines protocol header fields up to the transport layer and, thus, promises to be faster than Deep Packet Inspection (DPI)-based algorithms. The second algorithm is called Network Traffic Anomaly Detector (NETAD) [130]. It also analyzes parts of the application-layer payload for anomaly detection.

The modular architecture of Vermont allows to run every ADA on a single CPU core. All of the used ADAs in our framework are ordered as single pipeline stages in a sequential pipeline. As mentioned earlier, all algorithms are part of the controlled load allocation scheme of our framework.

4.2.2 Controlled Load Allocation Scheme

When combining multiple ADAs on one machine, performance bottlenecks have to be expected. The challenge is that different algorithms have different computational complexities and, thus, the packet throughput rates strongly differ. This requires that faster algorithms in the pipeline, should still be able to analyze packets although slower algorithms in front of them stall the pipeline. Thus, we implemented a controlled load allocation scheme. The goal of this scheme is to make sure that packets are analyzed by as many ADAs as possible. If there is no performance bottleneck, the load allocation scheme does not interfere.

Discarding packets if an ADA is overloaded is not a solution, as there might be algorithms later in the pipeline that could easily handle the packet rate. We solved this by using *sequential processing with packet skipping*. Here, packets skip an overloaded ADA and are immediately passed to the next algorithm without being tagged with an anomaly score. We refrained from using a window-based approach, because of possible negative implications if ADAs with similar performance demands are used. Instead, we use a probabilistic approach with our *feedback control algorithm*, which is invoked repeatedly. This algorithm determines the throughput quotas for every individual pipeline stage, based on the performance during a learning phase. These quotas are then used to determine a packet skip probability for every pipeline stage.

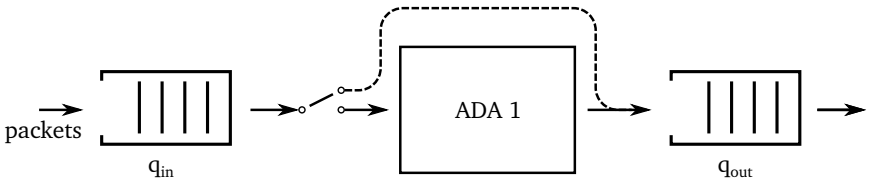


Figure 4.2 – Our controlled load allocation scheme: Packets skip the overloaded ADA with a certain probability until the algorithm recovers; from [23] ©2014 IEEE.

Figure 4.2 sketches the concept of skipping an overloaded algorithm instance. The pipeline architecture involves a queue in front and after every algorithm. This way, it is easy to detect a congestion (because of overloading) by observing the fill level of the single queues. We use the following *congestion criterion* to identify congestions:

$$d(q_{in}) > t_{in} \quad \wedge \quad d(q_{out}) < t_{out} \tag{4.1}$$

If the fill level of the input queue $d(q_{in})$ exceeds the threshold t_{in}, and the fill level of the output queue $d(q_{out})$ is below the threshold t_{out}, we define this pipeline stage to be congested. If a congestion is detected, the load allocation scheme controller throttles the packet rate for this algorithm by making packets skip this pipeline stage with the skip probability calculated by the feedback control algorithm. If the pipeline stage recovers (no congestion detected anymore), the load allocation scheme allows all incoming packets to be analyzed by this algorithm again.

4.2.3 Post-Processing of Packets

After the packets have been analyzed and tagged, we have different options for post-processing. One option is to filter out interesting anomalies or to report anomalies that match certain criteria. Even complete Flows containing such anomalous packets can be exported (possibly including payload information using Dialog-based Payload Aggregation (DPA)).

Another option is to use the build-in filtering engine to select packets based on the anomaly score. For example, the algorithms that we implemented as proof of concept generate score-based results indicating the anomaly level. Thus, in our evaluation scenarios, the filtering engine has been configured to only forward the most critical anomalies.

However, it is also possible to include algorithms that categorize packets. For example, an integer can be assigned to every single packet denoting the source network. In this case, the filtering engine can be configured to select packets matching a certain network.

To be able to analyze the behavior and assess the performance of the implemented algorithms, we added the possibility to collect statistics about the generated anomaly scores.

4.3 Evaluation

In this section we evaluate our proposed anomaly detection framework which combines multiple ADAs with the novel load allocation scheme. First, we assess the detection accuracy when using multiple detection algorithms. Then, we evaluate

the packet throughput capability of our system using the controlled load allocation scheme.

All the following experiments have been carried out using realistic traffic traces that we crafted the following way: The *anomaly trace* represents traffic from a typical home computer. It is composed mostly of internet traffic and some background traffic by the Operating System (OS) and installed applications. As this traffic does not contain any anomalies, we added 12 different types by modifying recorded packets and adding newly generated ones.

For the *attack trace* we concatenated traffic from the 1999 DARPA Intrusion Detection Data Sets [137]. We also included attack-free traffic to be able to train the semi-automatic algorithms. In total this traffic set contains 8 different types of attacks.

The *load allocation trace* has been created by capturing traffic from the uplink of a university campus network. The original data rate is 450 Mbit/s. This traffic is used to stress the implemented algorithms and assess the performance of the load allocation scheme.

The last trace is the *detection rate trace*. The trace consists of three categories of packets containing three different IP address ranges. The first category of packets, identified by the first IP address range, represents normal traffic. The second and third category represent anomaly traffic and contain exactly 500 packets each, which are equally distributed over the traffic trace (in 10 packets bursts). When replayed with the original packet rate (75 kpackets/s, 35 Mbit/s), the duration is 568 s.

4.3.1 Anomaly Detection Algorithms

In our first experiment, we assessed the detection accuracy of our framework using multiple ADAs. An analysis of the implemented algorithms is already provided in the corresponding publications. The goal of this experiment was to assess if the attack detection accuracy benefits from the combination of multiple ADAs.

We used a total of 3 algorithms: the semi-supervised PHAD algorithm, one instance of the NETAD algorithm in semi-supervised mode and another instance in the unsupervised mode. We used the build-in filtering mechanism and defined three sensitivity levels: A packet is considered anomalous only if it contains an anomaly score in the top 0.01% / 0.1% / 1% (low / medium / high). For this experiment we used the anomaly trace and the attack detection trace.

Figure 4.3 shows the results of the detection accuracy experiments. Using the anomaly trace (Figure 4.3a), a maximum of 12 anomalies can be detected and using the attack trace (Figure 4.3b) the maximum number is 8. As can be seen, with the used thresholds, not all anomalous packets passed the post-processing filter. No false positives are among the detected events. The graph clearly shows that, for

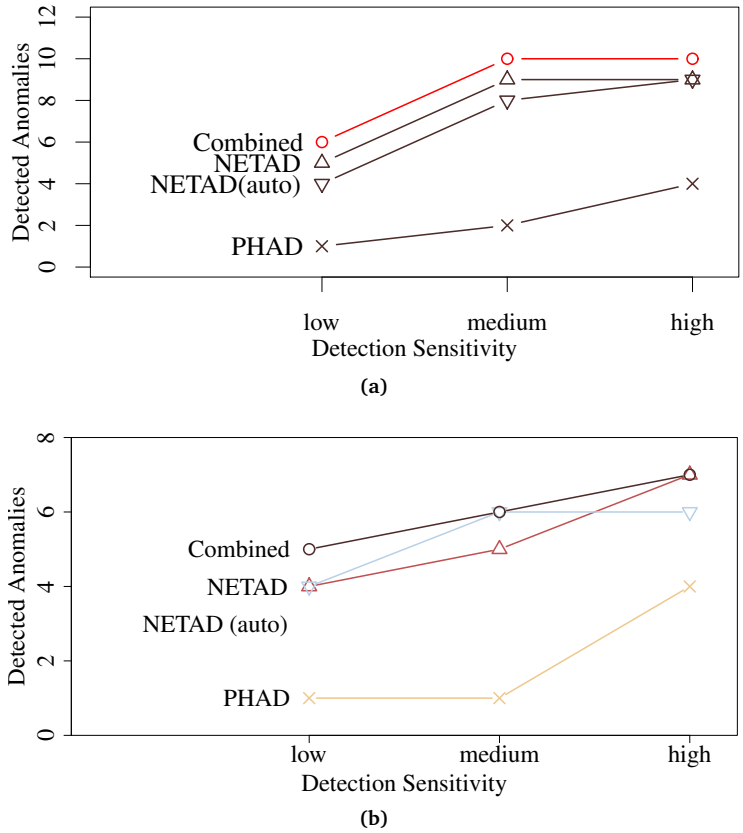


Figure 4.3 – Results of the detection accuracy experiment. (a) shows the results for the anomaly trace, (b) for the attack trace; derived from [23] ©2014 IEEE.

the anomaly trace, a combination of multiple algorithms increases the detection rate at all sensitivity levels. For the attack trace a combination of ADAs is most fruitful at low detection sensitivity. We conclude, that the detection performance can be increased already with a low number of combined algorithms, and we expect a further increase with more ADAs. Increasing the detection sensitivity would clearly increase the detection rate, but it would also increase the number of false-positive events, which is not desired in practice.

4.3.2 Controlled Load Allocation Scheme

The next experiment evaluated our controlled load allocation scheme. The key idea was to stress the implemented ADAs so that they cannot keep up with the packet rate, and therefore packets have to be skipped. Also for this evaluation, the anomaly detection framework has been configured to use the same three ADA instances as in the previous experiment. However, for this test we have used the load allocation

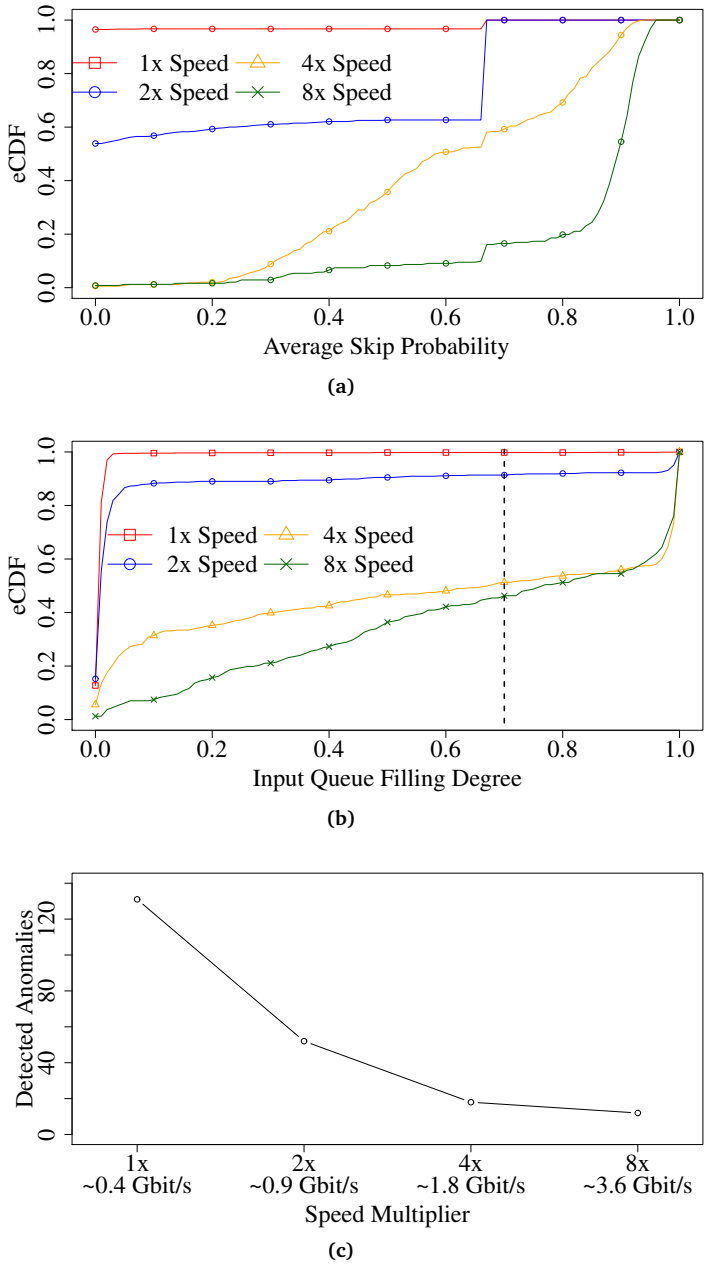


Figure 4.4 – Impact of packet rate on skip probability (a), input queue filling degree (b), and detection quality (c). The load allocation trace was used for this experiment; derived from [23] ©2014 IEEE.

trace and replayed it at different speeds in order to trigger congestions in the anomaly detection phase and, thus, being able to assess our controlled load allocation scheme.

The first experiment step evaluated the implemented feedback control algorithm, which is used to determine the skip probability for the single algorithm instances. The results are plotted as a CDF in Figure 4.4a and show the average skip probability for all three instances over four different packet throughput rates. If traffic is replayed at the lowest speed multiplier (1x, which corresponds to 450 Mbit/s), the average skip probability is almost always zero, which indicates that all of the single ADA instances can cope with the packet rate. Because of a singular glitch during packet replay, a packet burst beyond the defined replay rate occurred, which caused a temporary increase in skip probabilities. Looking at the skip probabilities for the speed multipliers 2, 4 and 8, we can observe that the skip probability increases with the throughput speed and, thus, works as expected. The high skip probabilities for multipliers 4 and 8 indicate that, at such high packet rates, the single algorithm instances are continuously overloaded.

The filling degree of the input queue for the above experiment is shown in Figure 4.4b. The threshold value used as congestion criterion is $t_{in} = 0.7$ and $t_{out} = 0.4$. With a speed multiplier of 1 there have been almost no congestions and, thus, $d(q_{in}) \leq 0.7$, which means that almost all packets could be analyzed by all algorithm instances. The congestion slightly increases with a speed multiplier of 2, and with speed multipliers 4 and 8, the fill level reaches the threshold values in more than 50 % of the measurement points.

Of course, also the number of detected anomalies suffers from packet skipping. Based on the input data, we have configured threshold values, so that only the most critical anomalies are reported if no packets are dropped. Therefore, all packets from the load allocation trace (which are about 285 million during a time interval of approx. 68 min) are processed.

Figure 4.4c shows the number of detected anomalies over the different speed multipliers. As expected, the overall detection performance decreases with increasing speed multipliers. For a speed multiplier of 2, only about 40 % of the anomalies could be detected. With the highest packet throughput (at 8x) only 12 out of 133 anomalies could be detected. This is in agreement with the skip probabilities for this replay rate, plotted in Figure 4.4a, where the skip probability is 80 % and higher in 80 % of the measurement points. Finally, we can see that the data burst, which caused the skip probability to rise for the speed multiplier 1x (as discussed above), did not have a significant influence on the detection rate.

4.3.3 Behavior under Stress

To get a better understanding of the behavior under heavy load we conducted the following experiment with the detection rate trace. As stated above, this trace contains normal packets and two times 500 anomalous packets, which have IP source addresses belonging to two address ranges. We implemented a new ADA, which assigns a high anomaly score if the incoming packet matches a given IP source address mask. Additionally, the CPU load of this algorithm can be manipulated by adding computational intensive calculations.

The experiment setup consisted of two instances of this ADA. The first instance was configured to detect the first 500 anomalous packets, by giving high anomaly scores to the first address range, and the second instance was configured to detect the second 500 anomalous packets. Additionally, we increased the CPU load (load value in the presented results) of the second ADA instance with every experiment run, to assess the impact on the detection performance. To be able to determine the influence of the load allocation scheme, we repeated the experiment twice. Once with the load allocation scheme turned on, and one run with the load allocation scheme turned off.

Figure 4.5 shows the number of detected anomalies with increasing load value for the second ADA instance. Without the load allocation scheme, the anomaly detection system drops substantially more packets, detecting less than 10 % at a load value level of 80 %. If our load allocation scheme is turned on, a significantly higher detection rate is achieved. Instead of random packet drops, which lead to both ADA instances not being able to analyze dropped packets, now, only the computationally more expensive ADA instance is skipped, while the other instance can still detect anomalies. This leads to a substantially higher detection rate: At a load level of 100 % almost 40 % of the anomalies can be detected.

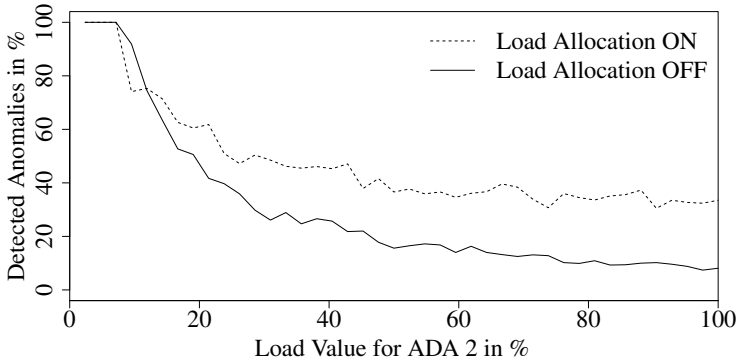


Figure 4.5 – Detection performance with increasing load on one ADA; derived from [23] ©2014 IEEE.

The experiment results underline that we we have successfully answered research question one: How can we combine multiple ADAs on a single machine, mitigating the negative impact of the high computational load, caused by multiple ADAs? We showed how to combine multiple ADAs on one machine and thereby increased the detection accuracy. Furthermore, we showed that our load allocation scheme is able to significantly reduce the negative impact of high computational load, caused by multiple ADAs on a single machine.

4.4 Lessons Learned

In this chapter an anomaly detection framework has been presented which optimizes attack detection by combining multiple Anomaly Detection Algorithms (ADAs) (of different types) on a single machine. The resulting high load is managed with our novel controlled load allocation scheme. We implemented this framework as part of the Free and Open Source Software (FOSS) Vermont network monitoring toolkit. The application scenario of our framework is high-speed networks, where it supplements traditional techniques for detecting novel attacks that have not yet been described for signature-based NIDS.

For the prototype shown here, we only implemented two different ADAs. Nevertheless, we were able to show that the detection performance can be increased by using multiple algorithms, and the high load can be mitigated with our load allocation scheme.

Chapter 5

Preprocessing HTTP for Network Monitoring and Intrusion Detection

In the previous chapter we proposed an approach to increase the efficiency of anomaly-based Network Intrusion Detection Systems (NIDS). In this chapter we propose novel methods for preprocessing Hypertext Transfer Protocol (HTTP) data before analysis for network monitoring appliances in general and NIDS in particular.

This chapter is based on the following publications:
F. Erlacher, W. Estgfaeller, and F. Dressler, "Improving Network Monitoring Through Aggregation of HTTP/1.1 Dialogs in IPFIX," in *41st IEEE Conference on Local Computer Networks (LCN 2016)*, Dubai, UAE: IEEE, Nov. 2016, pp. 543–546
F. Erlacher and F. Dressler, "High Performance Intrusion Detection Using HTTP-based Payload Aggregation," in *42nd IEEE Conference on Local Computer Networks (LCN 2017)*, Singapore: IEEE, Oct. 2017, pp. 418–425

5.1 Motivation

In recent years the portion of HTTP in internet traffic has increased continuously. By now it accounts for more than 50 % of the overall traffic volume and this portion is rising [2], [138]–[140]. There are several reasons for this: As pointed out in Chapter 3, more applications are moving from the desktop to the browser, which entails that more traffic will be transported over HTTP. Apart from that also lots of desktop applications use HTTP as a transport protocol, not least because it is usually not blocked by firewalls. And finally, the popularity of video streaming and its high-volume content also plays a major role in the rise of HTTP to the most used internet protocol.

As a consequence, it is required to pay closer attention to HTTP from a network monitoring perspective in general and an intrusion detection perspective in particular. Because of its widespread application, HTTP is increasingly used for malicious activity. This is emphasized by the high number of HTTP-related signatures for the popular NIDS Snort (cf. Section 5.2). Effective methods are needed to further analyze the carried HTTP payload. Because of the high data-volume, simply applying Deep Packet Inspection (DPI) techniques is not feasible in today's high-speed networks. This is also true for most other of today's modern and interleaving application-layer protocols. Because HTTP is the most widespread application-layer protocol, we apply our concepts to HTTP first, but they can be applied to most modern application-layer protocols.

In this chapter we propose two novel methods for preprocessing HTTP before analysis: Firstly, we extend the Internet Protocol Flow Information Export (IPFIX) protocol and include HTTP elements into IPFIX Flows as own Information Element (IE) fields. We introduce a mechanism that allows the aggregation of HTTP dialogs (request and response messages, which belong to each other) into bidirectional IPFIX

Flows (so called Biflows). This preserves the dialog-based nature of HTTP, which is especially relevant as the HTTP/2 protocol [69] shows that future application protocols will be much more interleaved [70]. The semantic correlation of HTTP request and response can be exploited, e.g., by NIDS, and is hardly possible to achieve with DPI-based methods. Because of the well structured nature of IPFIX Flows this allows for quick and easy analysis of HTTP internals.

Secondly, we propose a method to reduce the amount of incoming HTTP data for subsequent packet-based NIDS. The goal hereby is to reduce the amount of traffic data to analyze but retain all data relevant for intrusion detection. We filter the relevant parts for intrusion detection of HTTP along the lines of previous filtering solutions such as Time Machine [65] / Front Payload Aggregation (FPA) [66] and Dialog-based Payload Aggregation (DPA) [67]. The problem with these legacy approaches is that they base their filtering operations on transport layer flows. Thus, as explained in Section 2.1.2, they can not cope with the interleaving features of modern application layer protocols like HTTP/1.1 (e.g., pipelining of requests). This disqualifies them for modern, interleaving protocols (e.g., Dynamic Adaptive Streaming over HTTP (DASH) or HTTP/2). Our novel approach is called HTTP-based Payload Aggregation (HPA). It retains the important protocol parts by only forwarding the first N bytes of every message, being able to do so also for pipelined messages. Empirical studies [65]–[67] have shown that the first part of the payload is most relevant for attacks, while the rest is mostly insignificant for NIDS.

Both approaches have been implemented in the network monitoring toolkit Vermont (see also Section 2.4). This should lay the foundation for future network monitoring tools to efficiently analyze HTTP traffic.

The contributions of this work are the following:

- A methodology is presented for aggregating HTTP/1.1 dialogs into IPFIX Flow IEs.
- We initiated the standardization of our HTTP-related Flow IEs with the Internet Assigned Numbers Authority (IANA). They are now part of the official IPFIX standard.[20]
- We propose HPA, which applies HTTP-filtering for packet-based signature-based NIDS, reducing the exported HTTP payload to the first N bytes per dialog direction.
- Both concepts are thoroughly evaluated, comparing their functionality to related tools and assessing the network throughput performance.
- Our implementation is freely available as Open Source software `https://github.com/felixe/ccsVermont`, branch: http-aggregation.

[20] `https://www.iana.org/assignments/ipfix/ipfix.xhtml`

5.2 Importance of HTTP-Related Threats

To assess the importance of HTTP for intrusion detection, we analyzed the rule-sets for the NIDS Snort. We used the following databases for Snort signatures (as of 2017-01-15):

- All Snort rules (snapshot 2990) from Snort.org (for subscribers only).
- The Snort.org community rule-set.
- The Emerging Threats rule-set.[21]

When merging all of the above rules and removing duplicates, the following applies:

- 67 % (18363) of all active (uncommented) rules (27375) are related to HTTP (applying `HTTP_*` content modifier[22], applying the "service http" metadata tag or using the $HTTP_SERVER or $HTTP_PORTS variable).
- 66 % (12141) of these rules apply one of the `http_*` content modifiers; among these
- 94 % (11468) of these rules apply the pattern to a field in the HTTP header and, thus, the beginning of the HTTP message.

Overall, about 62 % of the HTTP rules apply a content search to the beginning of the HTTP message. This confirms:

- Firstly, most relevant intrusions nowadays are carried out using the HTTP protocol.
- Secondly, this confirms earlier findings, that the most relevant data portion for NIDS is located at the beginning of a Protocol Data Unit (PDU).

5.3 Aggregating HTTP into IPFIX

In this section we explain and evaluate our first approach, where we extend the IPFIX protocol and include HTTP elements into IPFIX Flows as own IE fields.

[21] `rules.emergingthreats.net/open/snort-2.9.0/emerging-all.rules`

[22] Content modifier in Snort rules reduce the pattern search in the content (payload) to only the portion defined in the modifier

5.3.1 Related Work In HTTP Monitoring and Aggregation

The following tools allow an in-depth investigation of HTTP: The popular packet analyzer *Wireshark* [141] includes an HTTP dissector which perfectly presents the single parts of an HTTP dialog. While Wireshark is the tool of choice for analyzing single HTTP connections, it is not designed for continuous monitoring of network traffic. *Zeek*[23] [19] on the other hand is a highly adaptable "network security monitor" which also allows continuous monitoring and dissection of network traffic in general and HTTP in particular, but also this tool fails for high-speed networks.

A common method to analyze the application-layer payload is to apply Regular Expressions (RegExes). This has been used to identify application-layer protocols in Transmission Control Protocol (TCP) (e.g., L7-filter [142]). The problem with this approach is its high computational cost; it is only applicable to low throughput traffic and, thus, not suited for continuous monitoring in high-speed networks [143].

To the best of our knowledge the only network monitoring tools that, at the time of publication, offer the possibility to export HTTP-related attributes using IPFIX are nProbe [144] and YAF [145]. nProbe is a network probe that was initially crafted to aggregate and export network data with the NetFlow [78] protocol but now supports also IPFIX. nProbe uses the PF_Ring [146] library providing high-speed network capturing. For nProbe, the exported IPFIX templates can be configured dynamically and support a few HTTP IEs if the HTTP plug-in is used.

YAF has been build as a reference implementation for IPFIX. By using the standard libpcap library its capturing speed is very limited. After configuring it to export a certain amount of payload, the DPI plugin takes care of the HTTP dissection. The DPI plugin checks the payload against a list of RegExes. This entails that YAF has no awareness of the HTTP message structure and, thus, applies RegEx patterns meant for the HTTP header also to the HTTP body. This is computationally very expensive and can lead to false positives, if the body carries a matching string (e.g., the RegEx searches for an HTTP method string which is present in the HTTP body). nProbe, in the version used in the evaluation, does not support the export of Biflows. YAF exports the whole HTTP dialog contained in a TCP flow in one IPFIX Flow and, thus, fails when advanced features like HTTP pipelining are used.

[23] `https://zeek.org`, formerly known as *Bro*

Table 5.1 – Comparison of features of IPFIX Exporters with HTTP capabilities.

	HTTP Biflow export	**Payload Aggregation**	**HTTP pipelining**
nProbe	two uniflows	no	no
YAF	one Flow	TCP only	no
Vermont	one Flow	TCP or HTTP	yes

All the mentioned tools including our novel implementation in Vermont are summarized in Table 5.1.

5.3.2 HTTP Aggregation Architecture

For our first approach, to enhance the IPFIX protocol and include HTTP elements into IPFIX Flows as own IE fields, we extend the IPFIX aggregation capabilities of the network monitoring toolkit Vermont (cf. Section 2.4). The module responsible for IPFIX aggregation is named *packetAggregator*. Our extension to this module includes TCP reassembly functionalities and HTTP protocol parsing capabilities. The workflow of the extended module is sketched in Figure 5.1. Incoming packets are first processed in the TCP reassembly engine which performs TCP connection handling and checks if the incoming packets can be processed or have to be queued. The reassembled TCP payload gets then analyzed by the HTTP parser which detects and exports HTTP dialogs. The aggregation into IPFIX IEs happens in all modules accordingly: The TCP reassembly engine aggregates all TCP related IEs, the HTTP parser aggregates HTTP-related IEs and all other fields are aggregated by the legacy packetAggregator functionality.

5.3.3 TCP Reassembly Engine

The TCP reassembly engine has to deal with a variety of challenges from handling (temporary) sequence gaps to maintaining connection state information and buffering out-of-order segments.

All TCP connections are represented by *TCPStream* objects and are maintained in a hash table. This table is indexed by a hash function (we use an XOR of the source IP, the destination IP, and the port number to make sure that both directions of a TCP connection end up in the same bucket). Per definition, the direction of the first packet of a connection is interpreted as the forward direction. This direction information

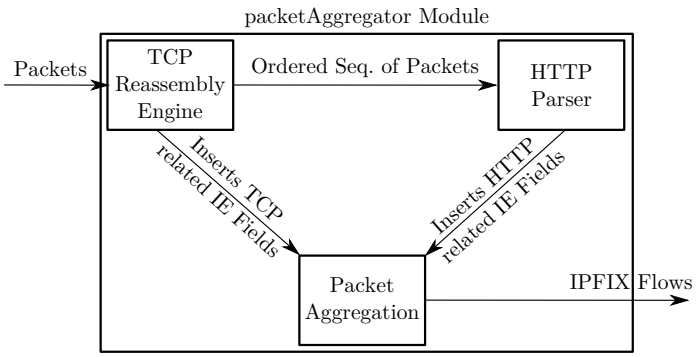


Figure 5.1 – Workflow of the extended packetAggregator module in Vermont.

is later needed for the HTTP parser to match HTTP requests/responses to dialogs. Every *TCPStream* object gets assigned a unique *TCP identifier*. Each connection also has a buffer to be able to deal with out-of-order or lost packets. Dharmapurikar and Paxson [147] evaluated the properties of these so called connection holes. They come to the conclusion that (i) most holes have a lifetime of less than 10 ms and (ii) nearly all holes require a buffer size of less than 10 kByte meaning that only a moderate impact on memory can be expected during TCP reassembly.

The TCP sequence number is used to keep track of the packet order. The TCP reassembly engine assigns four possible states to every connection. These states are shown in Table 5.2. The initial TCP state of a connection is TCP_UNDEFINED and is updated during the flow analysis. Normally, a connection goes through all states, but there are cases when single states might be skipped completely. For example, if handshake packets are lost, a connection can go from the TCP_UNDEFINED directly in the TCP_ESTABLISHED state.

To deal with orphaned connections, we introduced the timeouts summarized in Table 5.3. Each *TCPStream* object gets a time stamp assigned, which is refreshed with every packet arrival. Timeout checks are triggered whenever a packet arrives. To avoid checking all connections, the *TCPStream* entries are ordered oldest first. Starting the timeout checks with the first entry allows aborting the checks whenever a connection with a newer time stamp is found. The default values are rather short to keep memory usage low. To ease the use of trace files, the time stamp provided by the packet capturing library can be used instead of the real-time clock time stamp.

Table 5.2 – TCP connection states.

TCP connection state	Description
TCP_UNDEFINED	TCP connection state is not classified yet.
TCP_ATTEMPT	A packet with SYN flag was observed, the connection is currently in the phase of connection establishment.
TCP_ESTABLISHED	A TCP connection was observed, regardless if it was established regularly or not.
TCP_CLOSED	TCP connection was terminated by either a regular FIN sequence, a packet with RST flag, or a timeout.

Table 5.3 – TCP connection timeouts.

Timeout value	Default	Description
TIMEOUT_ATTEMPTED	5 s	The time after which an unfinished connection attempt expires.
TIMEOUT_ESTABLISHED	30 s	The time an established TCP connection may remain idle before expiring.
TIMEOUT_CLOSED	5 s	The time waited after a connection closes before expiring it. Useful to include packets which arrive with a slight delay.

This is useful when reading trace files at high speeds to preserve the original TCP connection behavior.

5.3.4 HTTP Parser

For performance reasons and to allow greater flexibility we did not use an existing HTTP parsing library but developed a so called *stateful on-the-fly HTTP parser*. Because the length of HTTP messages is unknown before parsing, we work on single incoming TCP segments. The segment's payload is parsed as much as possible, the rest of the payload is buffered and combined if the next segment of this TCP connection is available. Whenever a part of the HTTP message can be parsed successfully, the parsing state for the corresponding message is changed. This reduces the used buffer a lot in contrast to working on whole HTTP messages.

Based on the different parts of an HTTP message, we defined the following states (cf. Table 5.4): In the first step the HTTP parser checks the message type (request / response). Then, the parser extracts the necessary header field information and aggregates it into the corresponding IPFIX Flow field using so called enterprise specific IEs. If necessary, the parsing of the HTTP header can be configured to be skipped and the parser continues with the next step. Here, similar to the aforementioned Dialog-based Payload Aggregation (DPA) approach, the parser exports a configurable amount (namely the first N bytes) of data into the respective IPFIX IE. As soon as the message body is processed, the HTTP parser continues with the next HTTP message. If the type of the finished message is "response", it will be combined to an IPFIX Biflow with the corresponding HTTP request. Depending on the configuration, the parsing of the HTTP header can be skipped and only a configurable part of the message body is exported. For instance, this can be useful when the Flows are exported to a NIDS, which applies only rules inspecting the message body. To annotate issues encountered during the aggregation process, we implemented a dedicated IE.

Table 5.4 – HTTP message parsing states.

Parsing state	Description
NO_MESSAGE	HTTP message has not yet started.
MESSAGE_REQ_METHOD	HTTP request method was parsed successfully.
MESSAGE_REQ_URI	HTTP request URI was parsed successfully.
MESSAGE_REQ_VERSION	HTTP request version was parsed successfully.
MESSAGE_RES_VERSION	HTTP response version was parsed successfully.
MESSAGE_RES_CODE	HTTP response status code was parsed successfully.
MESSAGE_RES_PHRASE	HTTP response phrase was parsed successfully.
MESSAGE_HEADER	HTTP message header was parsed successfully.
MESSAGE_END	HTTP message was parsed successfully.

Typically, the full payload of the TCP segment is processed except for the following four cases:

1. Missing payload: a message element spans over two segments. The payload will be buffered until the next segment arrives.

2. Multiple messages: If HTTP pipelining is used, one segment may contain multiple messages (e.g., multiple GET requests). As each message has to be stored in a different Flow data structure the parser processes these messages individually.

3. Parsing failure: If, for any reason (e.g., data missing due to packet drops), the parser experiences a parsing error, then the rest of the payload is skipped and an annotation of the failure is added to the respective IPFIX Flow.

4. No HTTP content: If the segment's payload does not contain HTTP data, nothing will be aggregated.

The next parsing step is exporting the message body data into the respective IPFIX IE. Here, some heuristics help to speed-up the parsing process. Depending on the HTTP header, the parser determines one of the following message types and can thus decide how much payload it can skip:

1. No message body: If an HTTP status code 1xx, 204, or 304 is found, the HTTP message will not contain a message body.

2. Chunked transfer encoding: If the header field *Transfer-Encoding* is set to "chunked" the payload is split into several chunks of arbitrary size. But because each chunk is preceded by a chunk header defining its size, it is enough to parse the chunk header to be able to skip the rest of the chunk.

3. Fixed size: If the header field *Content-Length* is set to a fixed value, this amount of bytes can be skipped by the parser.

4. Multipart/byteranges: If the header field *Accept* defines a media type *multipart/byteranges* the message body length can be extracted from there. While this procedure is marked as deprecated in Fielding and Reschke [68], it is implemented for compatibility reasons.

Similar to the Dialog-based Payload Aggregation (DPA) approach mentioned in Section 2.1.2, Vermont allows to export the first N bytes of both directions of an HTTP dialog.

Table 5.5 lists the used HTTP-related IPFIX IE fields. Initially we used enterprise specific fields but decided later to register and standardize them with IANA. By now all but the flowAnnotation field are standardized. This should hopefully encourage

Table 5.5 – List of used IPFIX IEs; with IANA ElementID if standardized.

IE Name	Length	Description	ElementID
httpRequestMethod	16 Bytes	HTTP request method	459
httpRequestTarget	variable	HTTP response URI	461
httpMessageVersion	8 Bytes	HTTP request version identifier	462
httpRequestHost	variable	HTTP request host	460
httpStatusCode	2 Bytes	HTTP response status code	457
httpReasonPhrase	32 Bytes	HTTP response status phrase	470
flowAnnotation	4 Bytes	Annotations added to Flows during processing, e.g., parsing error, buffer overflow, TCP sequence gaps, and more	

manufactures of IPFIX probes and exporters to add these HTTP IEs to their IPFIX Flows.

5.3.5 HTTP Aggregation Evaluation

The evaluation of this part focuses mainly on functionality and throughput performance of Vermont with the improved packetAggregator module. For all of the evaluation experiments we used Vermont in the module configuration pictured in Figure 5.2.

In this configuration the *observer* module captures the packet from a packet source (Network Interface Controller (NIC) or stored packet trace). Because the *libpcap* packet capturing library used in the legacy observer module does not offer the possibility to capture packets at high rates, we changed the observer module to use the *PF_Ring* [146] library instead. Packets are then handed over to the enhanced packetAggregator module via a packet buffer of configurable size. The packetAggregator module aggregates the packets to IPFIX Flows as described in the previous section. The IPFIX Flows including the novel HTTP IEs are then handed over (via a Flow buffer of configurable size) to the *ipfixExporter* module which forwards the IPFIX Flows to the configured IPFIX Flow sink. The ipfixExporter module is included for exemplary reasons and is not strictly necessary for the experiments. Here, every module accepting IPFIX Flows could be used. The statistics presented in the following are all derived from the statistics engine shipped with Vermont.

The traffic traces used in the evaluation are either own, previously captured traces or traces which are obtained from public sources. We tried to have a representative and diverse set of realistic traffic traces which also includes possible border cases. For reproducibility reasons all traces (if not hindered by privacy issues) are published at `http://www.ccs-labs.org/~erlacher/resources/`.

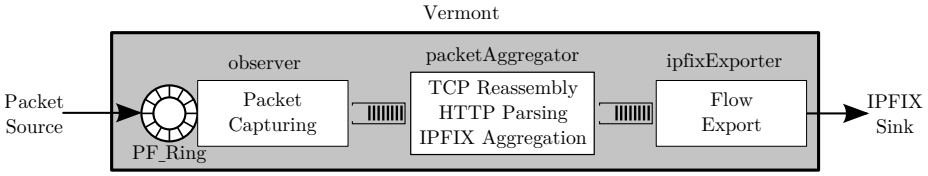


Figure 5.2 – Module configuration of Vermont for the aggregation and export of IPFIX including HTTP IEs.

5.3.5.1 Functional Evaluation of the TCP Reassembly Engine

To assess the functionality of our TCP reassembly engine we used different traffic traces, counted the TCP connection manually and then compared the outcome to the results of our parser and the results of following state-of-the-art tools: Wireshark (version 1.12), Zeek (2.2), nProbe (7.1), and YAF (2.7.1). Table 5.6 shows the detected number of TCP connections for all traffic traces and all tools.

All of the used traces are publicly available and taken from the study guide of the book "Wireshark Network Analysis" [141]. The trace names correspond to the names used in the book. The first trace (http-1) makes heavy use of the HEAD command. The second and third trace are visits to the www.espn.com website in 2011 and 2012 respectively, these traces contain many redirections. The fourth trace is a visit to the www.msnbc.com with unusual window scale factors.

The first row in Table 5.6 shows the manually counted and correct TCP connection numbers. As can be seen, Vermont, nProbe, and Wireshark count all TCP connections correctly. Zeek counts retransmitted TCP segments with SYN flag as own connections, this is the reason for the incorrect values but does not affect HTTP parsing. YAF seems to misinterpret packets of TCP connections that arrive after connection termination. For instance for some ACKs that arrive after the TCP FIN sequence a new connection is created.

While the values are not entirely in line for all tools, they are good enough to be used for HTTP parsing.

Table 5.6 – Number of TCP connections detected by the different tools, wrong results are marked in bold.

Tool \ Trace	http-1	http-espn2011	http-espn2012	http-msnbc
Correct Value	62	48	63	91
Zeek	**93**	**50**	**65**	91
Wireshark	62	48	63	91
nProbe	62	48	63	91
YAF	**94**	**52**	**67**	**92**
Vermont	62	48	63	91

5.3.5.2 Functional Evaluation of the HTTP Parsing Engine

To assess the correctness of the HTTP parser and the exported IPFIX Flows we first manually inspected the HTTP messages of all traces, and then compared the results again to the tools used in the previous section.

With Wireshark we applied the *Conversations* and the *HTTP-Load Distribution* function. We wrote a small script for Zeek in its own Domain Specific Language (DSL) to count the number of processed HTTP messages. For compatibility reasons with the other tools, we increased the TCP timeout values of Zeek from 5 s to 20 s, which are the same values used for Vermont. With nProbe we had to count the unique Flow identifiers which are given to the exported Flows when bidirectional export is enabled. To set up the export of HTTP information in YAF, we had to enable the HTTP plugin and turn *application label* support on.

Table 5.7 shows an exemplary fraction of the tested traffic traces and corresponding detected number of HTTP messages. More traces showed a similar behavior and are thus not shown here. What follows is a short description of the used and shown traces: The *riverbed-two* traces is a repeated visit to the www.riverbed.com web page. The peculiarity of this trace is that it does not contain any DNS queries and makes heavy use of the browser cache. The *cnn2012* trace contains visits to the www.cnn.com homepage and incorporates many TCP keepalive packets and TCP retransmissions. Both traces are taken from Chappell [141].

The *pipelining* trace contains traffic using the HTTP/1.1 pipelining feature. The *anomalous* trace is included to test the parser's robustness. In this trace, the structure of the included dialogs is valid, but some headers contain odd or unusual values.

Table 5.7 – Number of HTTP messages (requests and responses) detected by the different tools, wrong numbers are indicated in bold, MD = Matched Dialogs; from [24] ©2016 IEEE.

Tool \ Trace		riverbet-two	cnn2012	pipelining	anomalous
Correct	Req.	94	146	81	3966
Value	Res.	94	146	81	3966
Zeek	Req.	**54**	146	**8**	3966
	Res.	**54**	146	**8**	**3941**
Wireshark	Req.	94	146	**85**	**3816**
	Res.	**96**	**149**	81	**2907**
nProbe	Req.	94	**145**	**20**	**3965**
	Res.	94	**145**	**19**	**3875**
YAF	Req.	**14**	**131**	**4**	3966
	Res.	**9**	**145**	**0**	**4157**
Vermont	Req.	94	146	81	3966
	Res.	94	146	81	**3903**
	MD	94	146	**79**	**3102**

Some messages can possibly be interpreted as not ending regularly because the corresponding length fields contain wrong values.

In the following we explain the reasons for the differences in the results: Vermont and Wireshark are the only tools that wait for an HTTP message to end before increasing the message counter. This might lead to a higher number of messages compared to the result of the other tools. If Wireshark parses incomplete messages they are not counted but shown and tagged with “Continuation or non-HTTP traffic”. Wireshark counts HTTP messages with retransmitted TCP segments twice (or more). This should not be interpreted as an error but rather a design decision. Wireshark seems to have some problems with the anomalous traces, counting only 73.3 % of the messages right.

Zeek seems to require a complete TCP handshake to work correctly, this is the reason for the unusual low number of counted HTTP messages. On the other hand, the HTTP analyzer of Zeek seems to be the most robust, counting most of the messages of the anomalous trace right, compared to the other tools.

nProbe seems to prefer if HTTP messages are kept in one TCP segment: If HTTP requests are split over multiple segments, they are not detected. Two requests that are split over two HTTP messages are also not detected. If an HTTP request contains a very long Uniform Resource Identifier (URI), it is not recognized by nProbe. nProbe does not seem to support the HTTP/1.1 pipelining feature as it did not export the right values with the pipelining trace.

The wrong numbers of YAF are caused by the aforementioned RegEx application problem. For example, if YAF wants to detect HTTP responses it matches them only against the three digit response code followed by any text. If the message body contains a “404 Not Found”, the three digit RegEx will match again in the body and, thus, counts this response twice. In fact, for the anomalous trace YAF counts 4157 messages versus the 3966 manually counted ones.

Finally, our Vermont implementation counts almost all messages correctly. One inaccuracy happened in a trace not shown here: It counted an HTTP message twice because an IPFIX Flow timeout occurred. This underlines the importance of configuring the IPFIX timeout values according to the expected traffic. Other singular failures happened for small, partial messages. This only emphasizes the importance of a robust parsing engine which is able to deal also with border cases. Analyzing the exported IPFIX Biflows, Vermont matched the dialog pairs correctly in 99.7 % of the cases. Vermont only had minor issues with the anomalous trace. The only inaccuracies happened when there was a wrong message length information. If the length information was bigger than the actual size, Vermont counts the message as partial request, waiting for more data until a timeout occurs. This can be assumed as correct behavior. Vermont also seems to be the only tool keeping HTTP message

states: all other tools fail if the header exceeds the Maximum Transmission Unit (MTU) and stretches over two TCP segments.

5.3.5.3 Packet Throughput Performance Experiments

The test environment for the packet throughput performance experiments consisted of two workstations (Linux 3.13, i7-3920k CPU with 6 cores and 32 GByte RAM, Intel 82599ES NIC) which are interconnected with a 10 Gbit/s link. All experiments have been carried out by replaying a traffic trace from memory at Workstation A and capturing and aggregating the traffic with Vermont at Workstation B.

To replay the traces at a constant high rate, the program `pfsend` was used, which comes with the PF_Ring library. It proved to be much more reliable when replaying traffic traces at high-throughput rates than other programs like `tcpreplay`.

We configured Vermont the same way as in the functional evaluation experiments, exporting 2 kByte of HTTP payload in each IPFIX Flow. It has to be emphasized that exporting more payload only has a minor impact on performance.

To keep a reasonable long replay time despite high replay speeds, the traces used in this experiment are significantly bigger than in the functionality evaluation. The average number of packets of the traces is more than 12 million.

To stress the system as much as possible the traces contain HTTP traffic only. As in Section 5.3.5.2, only an exemplary fraction of the conducted experiments is discussed here.

Trace 1 was created by capturing the traffic going through an HTTP proxy used by a scientific work group for one week. We than removed messages with a size bigger than 5 MByte. This trace should represent typical internet browsing traffic. The next traces were created using a crawler and a limited MTU to assess the impact of the packet size on our implementation. *Trace 2* and *Trace 3* contain traffic with a maximum MTU of 1000 Byte. The traces differ in the number of TCP connections. *Trace 4* was captured with an MTU of 1500 Byte. During capturing of this trace the crawler came across very deep directory structures ending up with extremely long HTTP request headers and only very few different TCP connections.

Table 5.8 – General performance statistics of Vermont; from [24] ©2016 IEEE.

Trace ID	Pkt. Rate [k/s]	Through-put [Gbit/s]	Avg. Pkt. Size [Byte]	Avg. HTTP Header [Byte]	TCP Conn. [#]	Avg. HTTP Req. [$\times 10^3$/s]	Avg. HTTP Buffer [kByte/s]
1	810	4,13	675	475	17762	**29.78**	1540
2	790	2,50	375	448	1167631	**42.16**	481
3	730	4,41	732	488	8501	**32.03**	1
4	700	6,65	1161	**1880**	63	14.89	**27310**

The experiment results are shown in Table 5.8. The table is sorted by the packet throughput rate. For each trace the depicted rate was the highest rate possible without packet drops (or drop rates in the per mill range).

Looking at the numbers, we see that the packet rate depends mostly on the number of HTTP requests (Trace 2 & 3) and the size of HTTP request header. The reason for this is that a high number of HTTP requests require more effort by the HTTP parser, as well as more memory allocations because of a higher number of buckets for IPFIX aggregation. The size of the header also has an impact on the performance of the HTTP parser, because more data has to be processed and kept in memory before handed over to aggregation. This is especially true for Trace 4, where HTTP request headers are split over multiple segments and, thus, the parser has to wait until all segments are available before completing the request.

The memory consumption of the implementation is not problematic. It depends solely on the currently processed packets. The main bottleneck is the CPU performance which, in our experiments, reaches its limit long before memory is exhausted.

5.3.5.4 Impact of Packet Loss

With a stateful HTTP parser as in our case, a weak point by design is the parsing engine getting busy waiting for packets as soon as packet drops occur. The following experiment results show the behavior of our implementation in such a case. We replayed a network trace while increasing the packet rate drastically after about 130 s. Figure 5.3 shows the packet drops and the fill rate of the HTTP parsing buffer over time. As soon as the packet rate becomes too high the system has to drop packets. This means that message headers might not be parsed successfully, as it is not possible to determine the message end. Because these packets are lost completely, the parser has to scan the connection until it detects the beginning of the next message in order to terminate old, incomplete messages. Until that point in

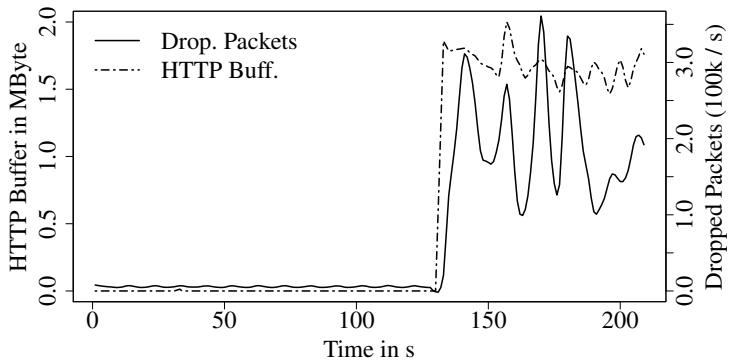


Figure 5.3 – Correlation of dropped packets and the HTTP buffer fill rate.

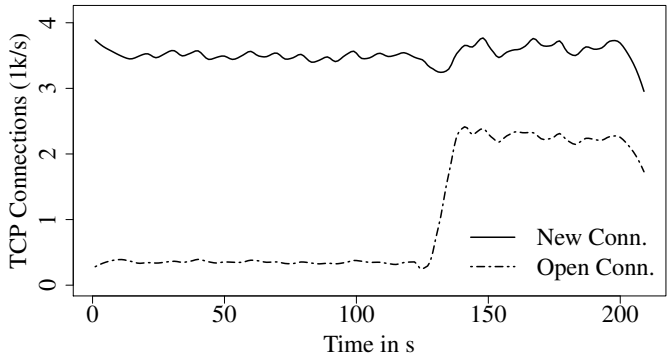


Figure 5.4 – Number of new TCP connections per second vs. open TCP connections. Packets start dropping at 130s.

time all unfinished HTTP messages have to be hold in the HTTP buffer, leading to an increase of the HTTP buffer rate.

The same holds for the TCP reassembly engine. Figure 5.4 depicts the behavior of the TCP engine in the same experiment. While the number of new connections per second stays the same, the rate of open TCP connections raises as soon as packet drops occur. As the reassembly algorithm assumes that lost TCP segments are retransmitted, it keeps the connection open, waiting for the lost segments. Because our implementation not only uses configurable timeouts but also detects possible ACKs for lost packets, it can close connections with lost packets also before a timeout occurs, saving a considerable amount of buffer space.

Both insights are proofs of the robustness of the implemented algorithms. Also when comparing the above throughput numbers with tests of HTTP parsers [148], Vermont places itself in the top range considering that it also performs TCP reassembly and IPFIX aggregation.

5.4 HPA: HTTP-Based Payload Aggregation

In this section we explain our second approach: our novel HTTP filtering method HPA. The main purpose of this filtering approach is to reduce the traffic data to be analyzed, as much as possible, while maintaining all detectable events. We export the filtered data in the form of packets, because this allows for seamless integration with existing intrusion detection solutions. Hereby we take advantage of the TCP-reassembly and HTTP-parsing capabilities of Vermont, which have been presented in the previous Section 5.3.2. Building on top of earlier work, we exploit the fact that most events detected by NIDS are located in the application-layer protocol header and / or at the start of an application-layer PDU as well as in the initial portion of the protocol payload.

5.4.1 HPA Concept

Because modern protocols do not follow the legacy concept of *'request followed by the corresponding response'*, simply using transport protocol sequence numbers to get insights into protocol details is not sufficient anymore. For example, if HTTP/1.1 pipelining is used, only looking at TCP sequence numbers to decode messages does not work. This demands a robust and stateful HTTP protocol parser that keeps track of multiple incoming TCP flows. *Robustness* applied to the HTTP protocol does not only include the ability to handle gaps caused by lost or dropped packets, but also being able to manage non-standard implementations used by current web servers and browser engines.

This entails that some steps like TCP reassembly and HTTP parsing will be done twice, once by our filtering implementation and once by the NIDS. While this is sufficient for our conceptual prototype, the computational overhead can be reduced in practical applications.

To show the advantages of our HPA concept, we compare it to the legacy filtering approach FPA and DPA, and, in Figure 5.5 reuse some of the sketches shown in Section 2.1.2. They show a comparison of filtering an exemplary HTTP connection with Time Machine / FPA and DPA as well as our novel filtering technique HPA. The baseline HTTP connection, as shown in Figure 5.5a contains two pipelined GET requests. A single line in the sketches represents a single HTTP message. All the messages use the same TCP connection.

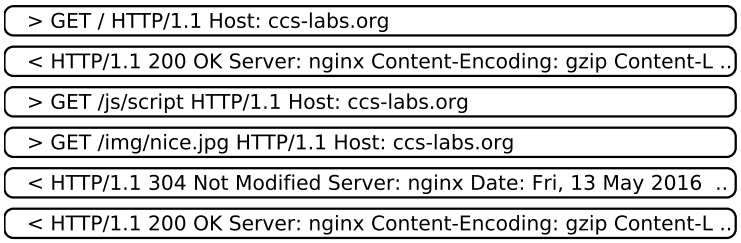
The FPA solution in Figure 5.5b retains the first N bytes of the outgoing and incoming direction of the TCP connection (after the handshake). It thus misses 4 HTTP messages.

The DPA approach in Figure 5.5c is aware of direction changes in the TCP connection and exports the first N bytes after every direction change. Therefore it is not able to deal with pipelined HTTP messages, that are issued following each other without waiting for the corresponding response.

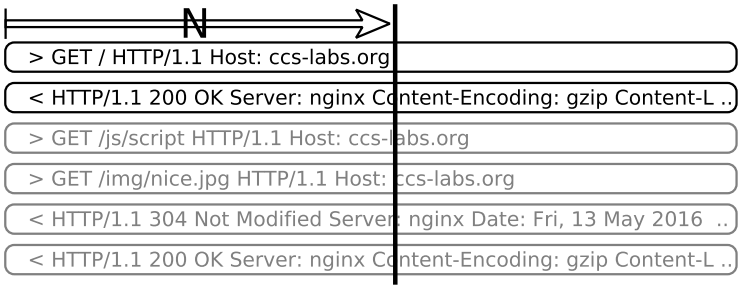
Finally, Figure 5.5d shows our HPA approach. Because it features a protocol specific parser, it is fully aware of the underlying states of the protocol and can export every single message. In the used example it is aware of the pipelined nature and is able to detect the corresponding request and responses as single messages, allowing to export and enable the NIDS to analyze only the first N bytes of all the contained HTTP messages.

5.4.2 HPA Implementation

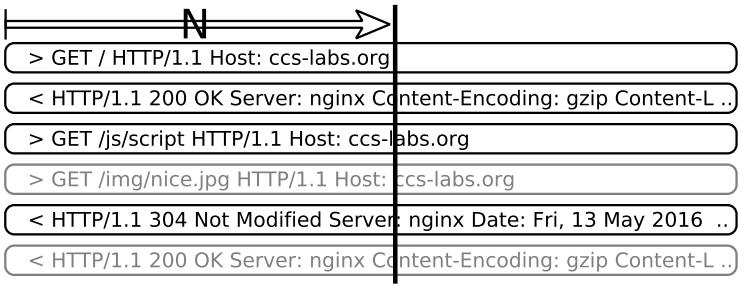
We use the capability of the HTTP aware *packetAggregator* module to export the first N bytes of every HTTP message and export this data portion as single packet per



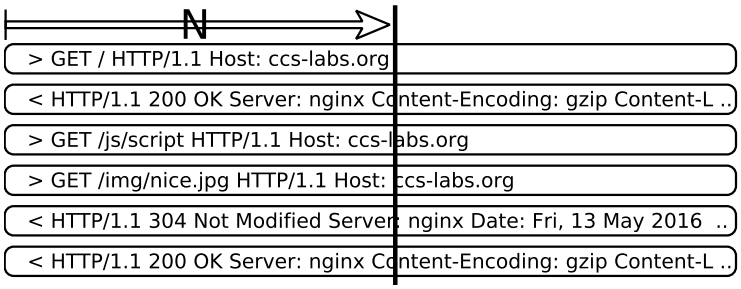
(a) Original HTTP connection



(b) Filtering using FPA



(c) Filtering using DPA



(d) Filtering using HPA

Figure 5.5 – Filtering of an HTTP connection with different techniques; from [25] ©2017 IEEE.

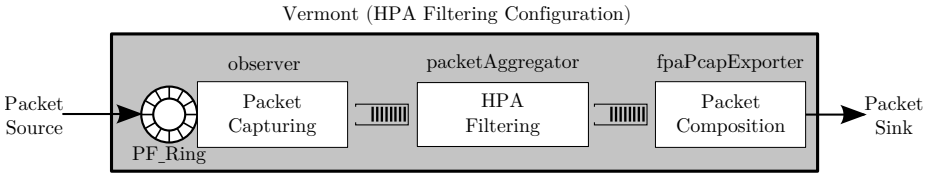


Figure 5.6 – Module configuration of Vermont for the HPA prototype.

message to the NIDS Snort. Figure 5.6 shows the Vermont module configuration for our HPA prototype.

The *observer* module fetches packets from a packet source (e.g., NIC or stored pcap trace). It already uses the PF_Ring enabled version of the observer module for high-speed packet capturing. Then the packets are forwarded via a packet buffer to the *packetAggregator* module. Here we take advantage of the enhanced version of this module, which includes a TCP reassembly engine and a stateful HTTP parser (cf. Figure 5.1). As explained earlier in this chapter, this module has been written for the export of HTTP-enriched IPFIX Flows. It exports one IPFIX Flow per HTTP message.

For HPA we take such an HTTP-enriched IPFIX Flow and convert it to a packet. For this, the IPFIX Flows are handed from the packetAggregator to the *fpaPcapExporter* module. This module was originally written for the FPA prototype, hence the name. For backwards compatibility we kept the name but the functionality has meanwhile been extended also for other filtering techniques. It takes the incoming IPFIX Flow and uses the HTTP payload to compose a packet. The number of bytes (N) to export for every HTTP message has thus to be stated in the packetAggregator configuration. This packet is then forwarded to the packet sink which, in our case, is the NIDS Snort.

5.4.3 HPA Evaluation

In the following we show the results of the evaluation experiments for our HPA approach. In Section 5.4.3.1 we show the results of experiments conducted to assess the correct behavior of the HPA prototype and test the influence of the filtering on the attack detection accuracy. We also compare the outcome to the experiment results of legacy filtering techniques. Finally, we assess the packet throughput capacity of our prototype and compare it again to the throughput capacity of legacy filtering techniques. The NIDS of choice is again Snort, because of its popularity and large sources of signatures. The structure of the prototype system is sketched in Figure 5.7. The used Vermont HPA filtering configuration is the same as explained earlier with Figure 5.6. After filtering the packets were handed to the NIDS Snort via a FIFO queue in the form of a named pipe. At startup Snort read the rules from a text file

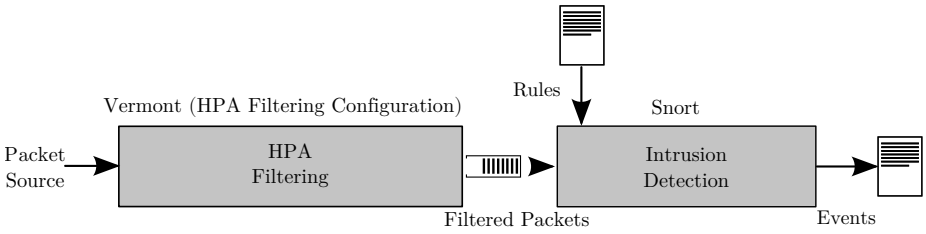


Figure 5.7 – Structure of the HPA prototype system including the NIDS Snort as used in the experiments.

and applied them during analysis to the incoming packets. If any rules matched, the corresponding event including information of the affected packet/stream was written to the event file.

5.4.3.1 Configuration and Traces Used for the Evaluation

The used Snort version was 2.9.9.0. With all of the following experiments, we run Snort in IDS mode with the default values as set in the snort.conf file, which is shipped with the installation archive. The only changes to these values are the following: Because of the high-throughput speed we increased the size of the queues in front of the detection engine (`max_queue_events` to 1000, `max_queue` to 1000, `log` to 1000) as well as the `max_queued_bytes` for the stream5_tcp preprocessor to 1.5 MByte. We also enabled the `-k none` switch to avoid Snort skipping packets with checksum errors. The applied rules are the 18363 active rules related to HTTP as described in Section 5.2. The way we fed packets to this system differed from experiment to experiment, but the used prototype is always the same.

Network traces (as far as privacy allows) and configuration files used in the evaluation part are available online.[24]

To assess the detection accuracy of HPA we need to show that events reported by Snort are not affected negatively by the filtering of the traffic.

As typical HTTP internet traffic we used a trace that we created by capturing the HTTP traffic going through a proxy of a scientific work group for one week. This is the same trace as used in the evaluation experiments in Section 5.3.5 and contains HTTP traffic only. For the rest of this chapter this trace is called *proxy trace*. To create additional malicious traffic, we created and inserted five prominent attacks to this trace. All five attacks are Common Vulnerability and Exposure (CVE)[25] registered in 2016 and have a relatively high Common Vulnerability Scoring System (CVSS) score. The patterns of the rules to detect these attacks are rather complex. Four of the patterns to detect are located in an HTTP response and one in an HTTP request. For each attack we inserted a normal and a pipelined version: The normal version

[24] `http://www.ccs-labs.org/~erlacher/resources/`
[25] `https://cve.mitre.org/`

was inserted in a legacy HTTP dialog containing one request and one response. The pipelined version is contained in an HTTP dialog where multiple requests follow each other without waiting for the corresponding response. In the four cases where the attack is located in the response, the first two pipelined requests ask for a 1024 Byte text file and only then issue a request for the attack response. Because the responses to pipelined requests have to be in the same order as the requests, this results in two 1024 Byte responses before the attack response arrives. For the single, handcrafted attack located in a request, we issued one pipelined request of 520 Byte, before the request with the attack pattern was issued. We inserted all five attacks 22 times (21 pipelined versions and 1 normal version). This results in 110 additional attacks added to the proxy trace. Finally, the proxy trace consists of more than 2.2 million packets and 2996 unique IP Flows (IP source / destination pairs). The average packet size of this trace is 825 Byte.

5.4.3.2 Functional Evaluation Experiments

In these experiments we made Vermont read the proxy trace directly from a file (cf. Figure 5.7). To be able to compare our HPA approach to legacy filtering approaches we also conducted experiments with FPA and DPA filtering which are also implemented in Vermont. To be able to assess the impact of the number of retained payload bytes, we repeated the experiments using different sizes of N. Finally, we compare the number of detected events in filtered traffic, with the events that Snort detected when directly reading the unfiltered traffic of the proxy trace. Table 5.9 shows the number of detected events by Snort for events already contained in the proxy trace. In the first column the rule SID (unique number identifying a signature) of the triggered event is listed. The second column shows the number of events detected when reading the unfiltered traffic of the proxy trace. The following columns show the number of detected events if traffic is filtered with HPA, DPA or FPA over different sizes of N (number of bytes of retained traffic), shown in the top row.

The numbers clearly show that HPA is able to detect most events, even when using a comparably small number of retained bytes of N=500 Byte. In this case, it even outperforms the DPA approach if DPA uses a larger N. This is especially true for rule SID 2013504. The data streams contained in the proxy trace may contain up to 10 pipelined requests. As explained in Section 5.4.1, in contrast to HPA, DPA and FPA miss patterns contained in pipelined requests and, thus, can not detect all events.

In the following we investigate some peculiarities in these numbers: For the first rule with SID 2013504 we see that more events were triggered if traffic was filtered with HPA than if Snort analyzed the unfiltered traffic. The reason for this is that Snort ignores TCP streams if they are terminated with a TCP RST packet. Our HPA

Table 5.9 – Number of Snort events with and without traffic filtering using the proxy trace; from [25] ©2017 IEEE.

		N=2000 Byte			N=1000 Byte			N=500 Byte		
Rule SID	Events without filtering	HPA	DPA	FPA	HPA	DPA	FPA	HPA	DPA	FPA
2013504	2035	2106	2041	937	2106	1497	519	2106	1022	272
40360	272	272	272	272	272	272	272	272	272	272
2012810	154	154	154	154	154	154	154	154	154	154
2015561	134	1	1	1	0	0	0	0	0	0
2101201	44	44	44	44	44	44	44	44	44	44
2001595	30	30	30	30	30	30	30	30	30	30
2014170	27	27	27	27	27	27	27	27	27	27
2101350	17	17	17	17	17	17	17	17	17	17
2013031	10	10	10	10	10	10	10	10	10	10
2018959	3	3	3	3	3	3	3	0	0	0
2016846	3	3	3	3	3	3	3	3	3	3
2012708	3	3	3	3	3	3	3	3	3	3
2011037	3	3	3	3	3	3	3	2	2	2
40158	1	0	0	0	0	0	0	0	0	0
2015707	1	0	0	0	0	0	0	0	0	0
2011507	1	0	0	0	0	0	0	0	0	0
Analyzed data in kByte	1878836	172601	170287	163725	89218	88027	84627	47527	46889	45084
in %	100	9.9	9.1	8.7	4.7	4.6	4.5	2.5	2.5	2.4

Table 5.10 – Number of Snort events for handcrafted attacks, with and without filtering using the proxy trace; derived from [25] ©2017 IEEE.

		N=2000 Byte			N=1000 Byte			N=500 Byte		
Rule SID	Events without filtering	HPA	DPA	FPA	HPA	DPA	FPA	HPA	DPA	FPA
40886	22	22	22	22	22	1	1	22	1	1
40778	22	22	22	22	22	1	1	22	1	1
37135	22	22	22	22	22	1	1	22	1	1
2023568	22	22	22	22	22	1	1	22	1	1
2022848	22	22	22	22	22	22	22	22	1	1

prototype, when exporting filtered traffic, will omit all TCP flags in packets. Snort thus analyzes the data contained in the filtered traffic in contrast to the unfiltered traffic containing TCP flags. If Snort analyzes the proxy trace with removed RST flags it detects the same number of events for this rule as with HPA filtered traffic. A negative effect of retaining only a portion of the traffic (by choosing a smaller number of N) can be seen for the rule with SID 2015561. If Snort analyzes the unfiltered traffic it detects 134 events, but as soon as traffic is filtered only 1 event is triggered. The reason for this is the following: In total five different TCP streams contain the patterns triggering this event. In one stream one of the patterns is located after 1050 Byte and, thus, detectable only in the case if N=2000 Byte. The other patterns triggering this event are located only after 9000 Byte or more. The same applies to the patterns triggered by the rule with SID 2018959 and the last four rules in Table 5.9.

Looking at the amount of data that Snort had to analyze, we see that with our novel HPA filtering approach the detection rate is higher with N=500 Byte and 47 MByte of Snort analyzed data vs. a worse detection rate of DPA with 170 MByte vs. the 1878 MByte for unfiltered traffic. This demonstrates that for intrusion detection on filtered HTTP traffic, HPA performs significantly better than legacy filtering methods.

The results for the 110 handcrafted events excluded from the previous table are shown in Table 5.10. As explained before, for each event, we inserted 1 version in a single request / response and 21 versions as pipelined HTTP messages after about 1500 Byte of data (1024 Byte HTTP body plus HTTP header). When traffic is filtered with DPA and FPA the events are only detected if more than 2000 Byte are retained. With N=1000 Byte and N=500 Byte and HPA filtering all events are detected. With DPA and FPA only the pattern of the non-pipelined attack can be found. These numbers evidence once more the problems that legacy filtering concepts have with interleaving protocols.

5.4.3.3 Network Throughput Performance

Filtering of traffic is only feasible if the filtering and the NIDS analysis of the filtered traffic are significantly faster than the analysis of the unfiltered traffic. To assess the network throughput performance of our HPA prototype we conducted the following experiment: We replayed the proxy trace at increasing packet rates between two workstations (Linux 3.2.0, i5-4440 CPU, 32 GByte RAM, Intel 82599 NICs) which were directly connected with a 10 Gbit/s network link. On the first workstation we used the `pfsend` program from the PF_Ring program suite [146], which proofed to be much more accurate when replaying traffic at high packet rates than other programs like `tcpreplay`. On the second workstation we used our filtering prototype

as configured in earlier experiments (cf. Figure 5.7), again applying the 18363 rules with Snort to unfiltered traffic and traffic filtered with HPA, DPA and FPA.

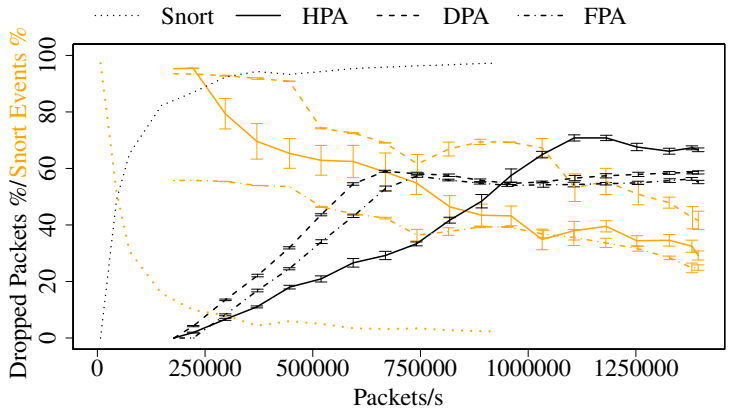
To be able to replay the proxy trace for the required time at high packet rates we concatenated this trace 15 times. We used `tcprewrite` to rewrite the IP addresses for every repetition in a deterministic way. This maintains TCP sessions between two hosts, but avoids that the exact same IP addresses are reused in repetitions of the same trace. The resulting trace contains 33 million packets and has a size of 28 GByte. The prototype setup was the same as used for the functional tests (cf. Figure 5.7), with the difference that this time we were capturing the network traffic from the NIC (instead of reading traffic from a trace). This realistic approach entails that if the filtering or NIDS analyzing stage cannot cope with the packet rate, packets will be tailbacked until they are dropped at the NIC and, thus, unavailable for intrusion detection. Spurious packet bursts, on the other hand, are absorbed by the buffers between the single stages. Again, we took advantage of the PF_Ring enabled Vermont observer module developed for the work presented in Chapter 5. Following Salah and Kahtani [149], to avoid performance losses due to context switches we set the Central Processing Unit (CPU) affinity of Snort to one dedicated CPU core and configured Vermont to use the other cores.

We replayed the proxy trace at increasing packet rates from 7500 packets/s (0.05 Gbit/s) up to 917 000 packets/s (6.2 Gbit/s) when analyzing the unfiltered traffic and from 178 000 packets/s (1.2 Gbit/s) up to 1.5 million packets/s (10 Gbit/s) when filtering the traffic before NIDS analysis with Snort. We repeated the experiment 10 times per measurement point.

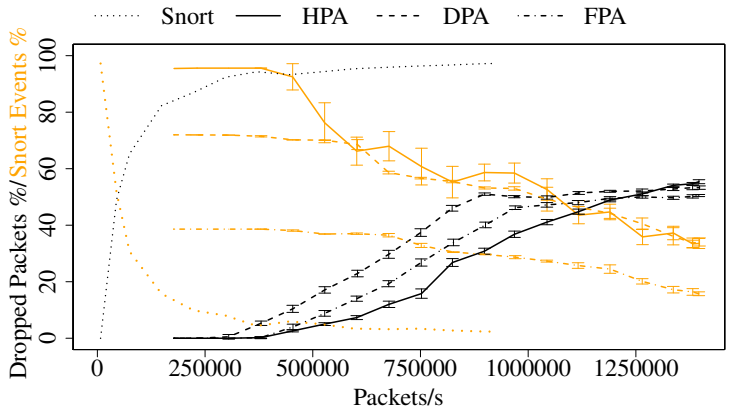
In Figure 5.8 the averaged results of the 10 experiment runs are shown. The data distribution between the single runs is shown with a confidence interval of 95%. The black lines show the number of dropped packets and the orange lines show the number of detected events. The dotted lines are the dropped packets percentage and detected events percentage of Snort analyzing the unfiltered traffic. The detected events of Snort when analyzing filtered traffic are shown with the other lines. The three subfigures a, b, and c show the different results when retaining a payload portion of N=2000 Byte, N=1000 Byte and N=500 Byte, respectively. The data for Snort is the same in all three subfigures but repeated for comparison reasons.

The detection rate, when looking at the numbers if no drops occur, is in line with the results from the functional evaluation in Section 5.4.3.2: For N=2000 Byte Snort detects most events, closely followed by DPA, followed by FPA with a detection rate of 60 %. For N=1000 Byte and N=500 Byte, HPA can keep its high detection rate, while DPA and FPA show a significantly worse performance.

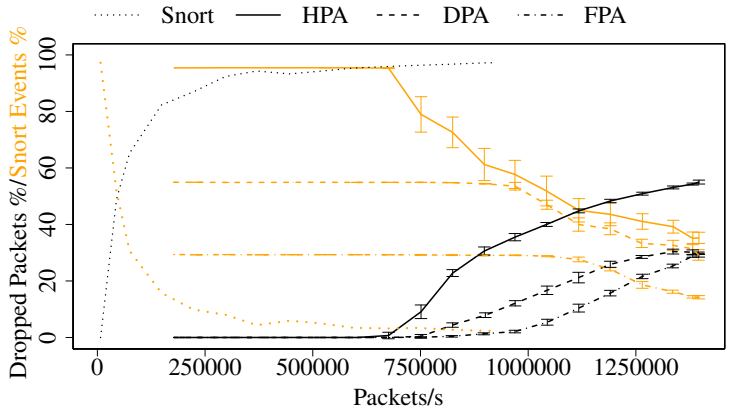
Snort, when analyzing the unfiltered traffic, can cope with a packet throughput rate of about 15 000 packets/s. At higher rates packets are dropped and the events



(a) N=2000 Byte



(b) N=1000 Byte



(c) N=500 Byte

Figure 5.8 – Detected events and dropped packets of the different filtering approaches over different packet rates, compared to Snort reading unfiltered traffic. Mean of 10 runs with confidence intervals (95%); derived from [25] ©2017 IEEE.

of these packets can not be detected anymore and, thus, also the detection rate decreases. Already at 230 000 packets/s the event detection rate drops below 10 %.

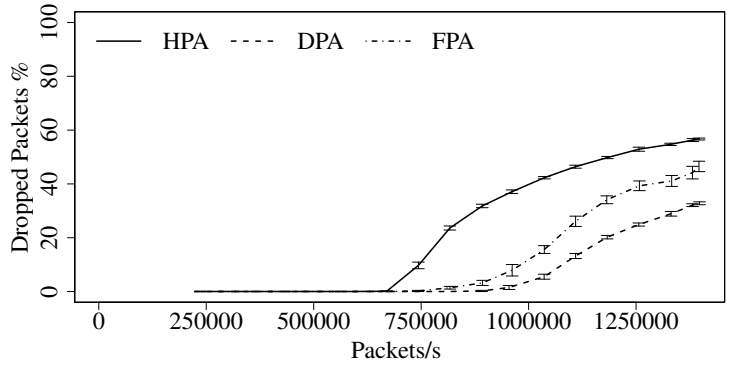
If Snort analyzes filtered traffic the packet rate without drops increases to 180 000 packets/s (1.2 Gbit/s) for N=2000 Byte and increases to 670 000 packets/s (4.4 Gbit/s) for N=500 Byte if our novel HPA filtering approach is applied. This corresponds to a speedup of a factor of 44, while retaining nearly 100 % of the events, compared to the packet rate without drops on unfiltered traffic. The packet rates of DPA and FPA are similarly high, but at the cost of significantly lower detection rates, especially at N=500 Byte. These data rates are higher compared to the data rates achieved in the original publication of DPA and FPA. This has to be attributed mainly to the use of the faster PF_Ring packet capturing library and to a smaller degree to the more modern workstations used for the experiment.

Generally speaking, our novel HPA filtering approach is computationally more expensive than DPA and FPA. But as we can see in the N=2000 Byte and N=1000 Byte plots, this does not mean it has to drop packets earlier. The reason for this is the very basic TCP reassembly strategy applied by DPA and FPA. These filtering approaches use TCP sequence numbers for TCP reassembly and do not keep states for TCP connections. This way, data from bypassing packets is stored at the location dictated by the sequence number. If packets are dropped, empty spots in the TCP connection are filled with binary zeros, which are still present in the packets exported to the NIDS. This means in our case that Snort analyzes these zeroed out parts of the TCP connection which costs additional CPU cycles. By using a stateful TCP reassembly engine and a stateful HTTP parser, our HPA prototype avoids this behavior and Snort does not have to waste precious CPU cycles to analyze zeroed data.

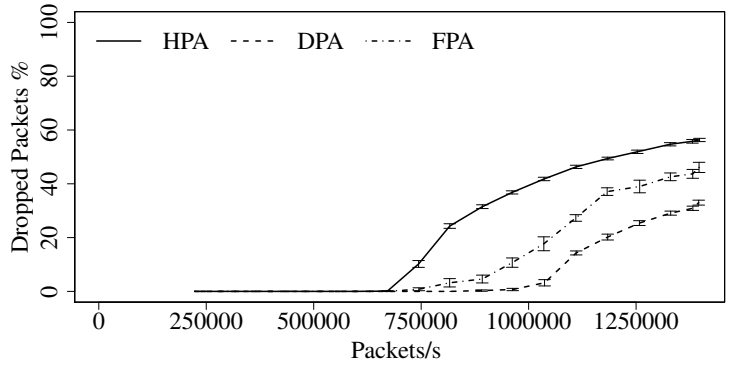
Another peculiarity shown in these plots is that the rate of dropped packets increases fairly fast, while the detected events do not drop at the same ratio. The reason for this is that packet drops are usually distributed rather uniformly. However, as stated multiple times earlier in this thesis, data relevant for intrusion detection is mostly found at the beginning of a PDU. This means that the drop probability of a packet containing intrusion detection relevant data is lower than the overall packet drop probability.

As the numbers of the functional and performance tests reveal, the event detection rate of HPA is only to a small degree dependent on the size of N, while DPA and FPA require a large N to achieve a high event detection rate. Therefore, the rate of HPA N=500 Byte should in fact be compared to the results of DPA and FPA at N=2000 Byte. Thus, HPA has a performance increase of a factor of more than 3 compared to DPA.

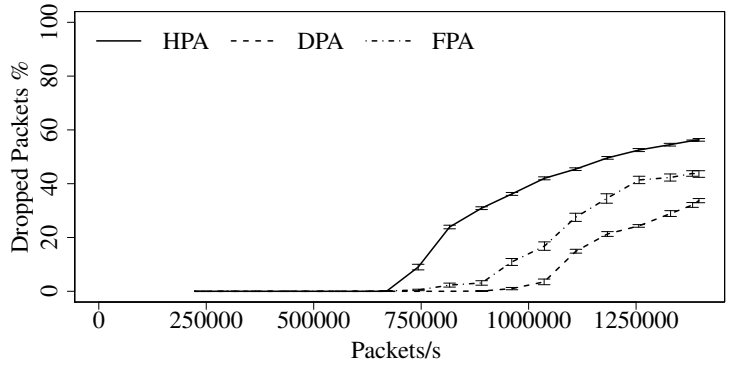
Because the experiment results shown above might leave the suspicion that the bottleneck might be the NIDS and not the filtering engine, we conducted a final experiment, assessing the packet throughput rate of the filtering prototype



(a) N=2000 Byte



(b) N=1000 Byte



(c) N=500 Byte

Figure 5.9 – Speed of Vermont filtering the traffic and exporting packets to a RAM disk instead of to a NIDS. Mean of 10 runs with confidence intervals (95%); derived from [25] ©2017 IEEE.

alone. The experiment was exactly the same as above, with the difference that this time, instead of exporting the packets to Snort via a named pipe, we configured Vermont to export the packets to `/dev/null`. Figure 5.9 shows the average of 10 experiment runs for every measurement point with 95 % confidence intervals. Comparing the packet throughput rates with the previous experiment results shown in Figure 5.8 we can see, that especially for N=2000 Byte and N=1000 Byte the rate is substantially higher now. This means that, when Snort had to analyze more data, packets were tailbacked. This also entails that the overall packet throughput rate could be increased by a faster NIDS.

The numbers also evidence, that without subsequent analysis, the advantage of the computationally more expensive stateful parsing approach of HPA is not taken advantage of and, thus, HPA shows a reduced performance compared to DPA and FPA. Nevertheless, it has to be emphasized that, as shown in Figure 5.8, despite the slightly lower packet throughput rate of HPA it retains a substantially higher event detection rate.

5.5 Lessons Learned

In this chapter we proposed two novel methods for preprocessing HTTP before analysis. We introduced the aggregation of HTTP dialogs into bidirectional IPFIX Flow records. As we will see in the following chapters, this allows for easy export to other network monitoring appliances. The implementation in the network monitoring toolkit Vermont aggregates and exports HTTP header fields, as well as a configurable amount of bytes of the HTTP payload into a set of IPFIX IE fields. The aggregated form of the important parts of HTTP allows for a quick and efficient analysis in following network analysis tools. In the evaluation we showed that our HTTP parser outperforms other state-of-the-art network monitoring tools in terms of correctly detected HTTP messages. The implemented parser proved to handle all kinds of HTTP traffic, even complex behavior such as HTTP pipelining. Our performance experiment results show that our framework can cope with packet throughput rates of 800 kpackets/s and with data rates of 6 Gbit/s and, thus, creates the potential to monitor HTTP even in multi-gigabit networks.

Furthermore, we presented the HPA traffic filtering approach, which increases the performance of packet-based NIDS by significantly reducing the amount of HTTP traffic, while retaining almost all data relevant for intrusion detection. HPA, similar to legacy filtering techniques such as Time Machine / FPA, and DPA exploits the heavy tailed nature of internet traffic. We extended this concept to modern and interleaving internet application protocols and developed a prototype for the Hypertext Transfer Protocol (HTTP) version 1.1. HPA is able to capture the first N bytes of every

HTTP request and response, even if advanced features like pipelining are used. We conducted evaluation experiments using realistic traffic and exporting the filtered traffic to the NIDS Snort. We also compared the results of our HPA approach to the legacy filtering methods DPA and FPA. Snort analyzing the traffic filtered by HPA, shows a speedup of a factor of 44 compared to the packet throughput rate of Snort analyzing unfiltered traffic. The experiments also show, that the HPA filtering approach outperforms other filtering methods by exhibiting an event detection rate of more than 97 %, with only 2.5 % of the network traffic to be analyzed.

This chapter answers research question two: How to reduce the amount and aggregate the interesting parts of HTTP traffic for network monitoring and intrusion detection. We accomplished this by exporting the important parts of the HTTP header and a definable amount of data from the beginning of the HTTP payload. Additionally, the standardization of these IE fields by IANA guarantees that standard compliant network monitoring appliances can analyze these fields without further processing.

Chapter 6

FIXIDS: A Signature-Based Flow Intrusion Detection System

In the previous chapter we proposed novel methods for preprocessing Hypertext Transfer Protocol (HTTP) data before analysis for network monitoring appliances in general and Network Intrusion Detection Systems (NIDS) in particular. In this chapter we propose a novel, signature-based NIDS, which uses the HTTP-enriched Internet Protocol Flow Information Export (IPFIX) Flows proposed in the previous chapter for intrusion detection analysis.

This chapter is based on the following publications:
F. Erlacher and F. Dressler, "FIXIDS: A High-Speed Signature-based Flow Intrusion Detection System," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2018)*, Taipei, Taiwan: IEEE, Apr. 2018
F. Erlacher and F. Dressler, "On High-Speed Flow-based Intrusion Detection using Snort-compatible Signatures," *IEEE Transactions on Dependable and Secure Computing*, submitted

6.1 Motivation

As explained in Section 1.1, signature-based NIDS analyze network traffic for malicious activity by searching for patterns of attack signatures in the received packets and streams. These pattern-matching operations are very performance intensive, and the precise attack detection capability comes at the cost of relatively low traffic throughput performance. Accordingly, there is the need for novel, more efficient approaches to signature-based intrusion detection.

We explained in Section 2.2.1, that a common approach for analysis in high-speed networks in general and to some extend also for intrusion detection in high-speed networks, is Flow monitoring. The most used standard for the exchange of Flow information is the IPFIX protocol [72], [74], [81]. In Section 2.3 we presented the current state-of-the-art in Flow-based intrusion detection, which consists almost exclusively of anomaly-based approaches. This is owed to the fact that Flows contained only header-based informations and no application-layer payload. Thus, signatures, which usually include patterns contained in the application-layer payload, can not by applied.

Therefore, in Chapter 5 we took HTTP as an example for a modern interleaving protocol used by internet applications and implemented a network monitoring tool, which is able to export the most important parts of HTTP as IPFIX Information Elements (IEs). These HTTP-related IEs have meanwhile been standardized by the Internet Assigned Numbers Authority (IANA).[26] Manufacturers of commercial and open-source network appliances have also started to include HTTP IEs into their

[26] `https://www.iana.org/assignments/ipfix/ipfix.xhtml`

set of exportable Flow fields. Examples are Citrix,[27] Sonicwall,[28] Vermont,[29] and Ntop.[30] Most of the manufacturers have started exporting HTTP-related IEs before or during our standardization efforts, which means that not all where able to consider the standardized fields yet. But, as we will see later in this chapter, the format of the used fields is basically the same.

In this chapter we take advantage of this novel concept of payload-based fields in Flows and use them for signature-based intrusion detection. We call our prototype IPFIX-based Signature-based Intrusion Detection System (FIXIDS). FIXIDS uses HTTP-related signatures of the well known NIDS Snort and applies them to the aforementioned IPFIX Flows containing HTTP-related IE fields. To the best of our knowledge, this is the first time that HTTP IEs are directly exploited for signature-based intrusion detection. By using the IANA standardized IPFIX HTTP IEs, we assure that FIXIDS can accept Flows from every Flow Exporter supporting these IEs. With the use of Snort signatures we make sure that the system has thousands of precise, up-to-date and community validated attack descriptions available. When compared to traditional Deep Packet Inspection (DPI)-based intrusion detection systems like Snort, the Flow-based intrusion detection approach has the additional advantage of separating the task of traffic parsing and Flow aggregation from the analysis stage and, thus, makes it possible to distribute both stages on different machines. This is not possible with legacy NIDS like Snort, where the traffic parsing and decoding has to be done in a tightly coupled fashion with the analysis stage.

We want to emphasize that FIXIDS is not intended as a replacement for traditional NIDS. The concept of our approach is destined for campus or corporate high-speed networks, where traffic is aggregated to IPFIX Flows by in-situ appliances like IPFIX capable switches. Here, FIXIDS can be applied to remove a substantial part of the load of traditional NIDS like Snort (cf. Section 6.5.3).

As can be seen in the evaluation part of this chapter, the main advantage of FIXIDS is that it can analyze HTTP traffic significantly faster than Snort without losing detection accuracy.

The main contributions of this chapter are the following:

- We present FIXIDS, a novel concept for signature-based intrusion detection on IPFIX Flows.

- FIXIDS takes as attack description HTTP-related signatures from the widespread NIDS Snort, but it is also possible to manually define attacks.

[27] https://www.citrix.com/products/netscaler-adc/netscaler-data-sheet.html
[28] https://www.sonicwall.com/en-us/products/firewalls/management-and-reporting/global-management-system
[29] https://github.com/felixe/ccsVermont
[30] http://www.ntop.org/products/netflow/nbox/

- We thoroughly evaluated FIXIDS with the help of the realistic traffic generator TRex and the attack traffic generator GENESIDS.

6.2 IPFIX-based Signature-based Intrusion Detection System (FIXIDS)

The main goal of FIXIDS is signature-based intrusion detection for high-throughput speeds without loosing event detection accuracy. Our approach is to take advantage of the aggregated and structured nature of IPFIX Flows. By using HTTP-related IE fields, FIXIDS can precisely detect attacks and at the same time keep the amount of analyzed data low, in comparison to traditional intrusion detection on packets.

6.2.1 Rules and Signatures

To take advantage of thousands of up-to-date and community validated signatures, we decided to take the same rule syntax as Snort. Because of the usage of HTTP-related IEs, FIXIDS supports Snort attack signatures for HTTP traffic only.

The main component of a Snort signature are content patterns to be searched in the payload (in our case in the IEs) of the traffic to analyze. To speed up the pattern-matching process Snort allows to narrow the search space by so called *content modifiers*. In the case of HTTP content modifiers it restricts the search space to single fields, like request method or request Uniform Resource Identifier (URI). The content modifiers supported by FIXIDS are shown in Table 6.1, together with the related IPFIX IE as well as the official IANA IE ID. Additionally to the shown keywords, also the “uricontent” keyword is supported. It is semantically equivalent to a content keyword restricted with the http_uri content modifier. Also the “nocase” modifier can be additionally applied to a content pattern and enables case insensitive text pattern-matching.

FIXIDS supports text and binary data (hexadecimal representation), as well as content pattern description as Perl Compatible Regular Expressions (PCREs). Also

Table 6.1 – Snort content modifiers supported by FIXIDS and their corresponding IPFIX IE name and IANA ID; derived from [26] ©2018 IEEE.

Content modifier		HTTP IE	IANA IE ID
http_method	→	httpRequestMethod	459
http_uri	→	httpRequestTarget	461
http_raw_uri	→	httpRequestTarget	461
http_stat_code	→	httpStatusCode	457
http_stat_msg	→	httpReasonPhrase	470

for PCRE patterns content modifiers are allowed. They immediately follow the PCRE pattern and are listed in Table 6.2 with the corresponding behavior.

It goes without saying that the Snort rule syntax can also be used to manually create own rules, in cases an attack has not been described yet in the Snort data bases or, to simply generate customized signatures for specific application scenarios.

6.2.2 Implementation

FIXIDS is implemented as an own module named *ipfixIds* in the network monitoring toolkit Vermont (cf. Section 2.4) and as such licensed under a GNU General Public License (GPL) and freely available.[31] A typical configuration of Vermont including the FIXIDS module is sketched in Figure 6.1.

FIXIDS can receive IPFIX Flows from switches or any other IPFIX exporter sending standard compliant Flows. The module responsible for receiving the Flows (ipfixCollector) currently supports the following transport protocols: TCP, UDP, DTLS over UDP and SCTP. Options, e.g port number or other protocol specific preferences can be set in the Vermont configuration file. After the Flows are received by the ipfixCollecotr module, they are handed over (via a buffer of configurable size) to the ipfixIds module which is responsible of conducting the intrusion detection analysis.

[31] `https://github.com/felixe/ccsVermont`

Table 6.2 – Modifiers for PCRE content patterns supported by FIXIDS.

Modifier	Description
i	Performs case insensitive pattern matching
U, I	The PCRE pattern matching is applied to httpRequestTarget IE
M	Pattern matching is applied to httpRequestMethod IE
S	Pattern matching is applied to httpStatusCode IE
Y	Pattern matching is applied to httpReasonPhrase IE

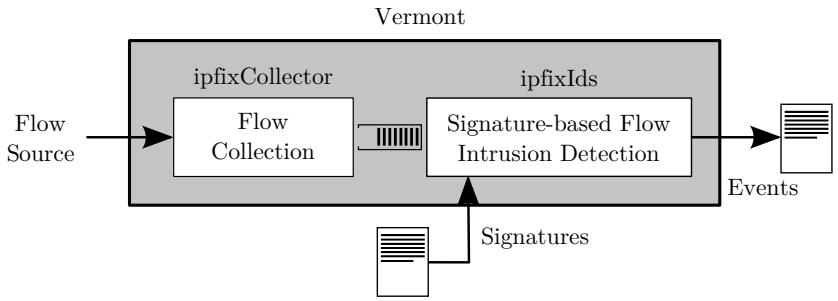


Figure 6.1 – Minimal Vermont configuration with FIXIDS functionality.

The signatures used as attack definitions for intrusion detection are parsed at startup and from then on kept in memory. During analysis this module matches the HTTP IEs of IPFIX Flows to the corresponding patterns of all signatures. This pattern-matching analysis is done in parallel as sketched in Figure 6.2. The number (*n*) of used pattern-matching threads is stated in the module configuration and parsed at startup. Typically, this number is adapted to the available cores of the machine running FIXIDS. During runtime, incoming Flows are distributed in a round-robin fashion to *n* FIFO queues (*n* being the number of pattern-matching threads).

The pattern-matching threads use the dedicated Flow queue as input and continuously remove a Flow and match the contained IEs to the patterns of all rules. Because all HTTP IEs of interest are UTF encoded, we use a string-compare function to match IEs to string patterns of signatures. If a pattern contains a character in hexadecimal representation it is converted to its ASCII equivalent before comparison. The *strstr()* C string-compare function (*strcasestr()* for case insensitive search) has proven to show the best performance for our application scenario. I takes advantage of hardware instructions to make direct usage of Central Processing Unit (CPU) registers.

If signature patterns are represented using a PCRE, we compare it to the IE string using the *boost::regex* library.[32] Signatures usually contain more than one pattern. Typically, there is an initial, easy to match pattern like the request method. Same as other NIDS, we take advantage of this and abort the pattern matching for a specific Flow as soon as one pattern match fails. An alarm is triggered only if all patterns match, in this case the event information is written to the event file of this thread and the analysis is continued with the remaining signatures.

To avoid race conditions or expensive access control mechanisms we configured every pattern-matching thread to use the same Flow input queue and the same

[32] `https://www.boost.org/doc/libs/1_62_0/libs/regex`

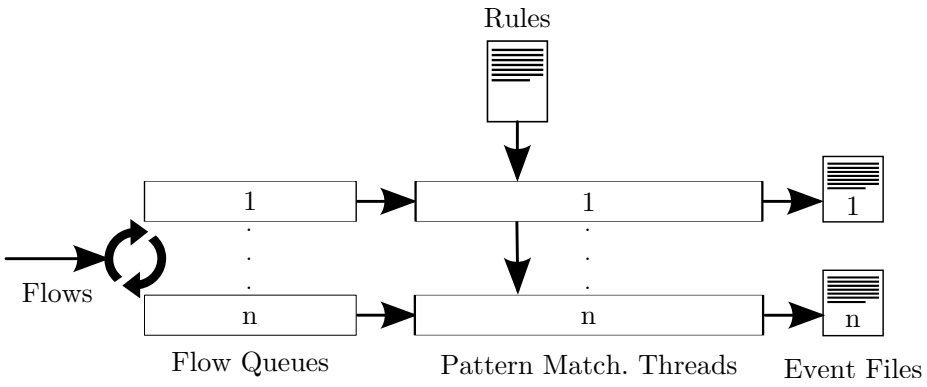


Figure 6.2 – Sketch of the internals of the FIXIDS module.

event file over the whole runtime. To achieve the same format for the event file as single-threaded FIXIDS the individual event files can be concatenated and sorted.

Because of the modular nature of Vermont the configuration can be changed to take as input live traffic from a Network Interface Controller (NIC) or a captured network trace file. In this case, an additional module called *packetAggregator* will aggregate the packets to IPFIX Flows before intrusion detection analysis.

6.3 Evaluation Experiment Setup

In this section we describe the tools and their configuration that we used for our evaluation experiments. To allow reproducibility we published all configuration files and other resources.[33]

6.3.1 Snort

Because Snort is the most widely deployed NIDS, we use it to get a baseline both in detection accuracy and packet throughput performance to compare to. In Section 6.4 we compare the detected events of Snort to the detected events of FIXIDS. We want to emphasize that we do not investigate if the events produced by Snort are accurate or not. We assume that this has been done by the members of the Snort community, which analyze and inspect the Snort signatures in the rule databases. For the evaluation we compiled Snort (version 2.9.11.1) from scratch. We use the default snort.conf configuration file and run Snort in IDS mode. The only changes to default values are the following: To make sure all events are reported (same as in FIXIDS), even at high-throughput speeds, we increased the size of the queues in front of the detection engine: max_queue_events: 1000, max_queue: 1000 and log: 1000. For the same reason we had to adapt the memory limits of the TCP reassembly engine (named stream5): memcap: 512 MByte, max_queued_bytes: 128 MByte and max_queued_segs: 20000. Again, we used the *-k none* switch to make Snort accept packets with checksum errors.

6.3.2 FIXIDS Setup

With FIXIDS we refer to the Vermont configuration as sketched in Figure 6.1 and described in Section 6.2.2. The transport protocol used to carry IPFIX Flows is dependent on the Flow Exporter and denoted in the description of the specific Flow exporting tool. For experiments where we used a slightly different configuration as compared to the one above, we make this clear in the experiment description. In all

[33] `http://www.ccs-labs.org/~erlacher/resources`

of the following experiments we used four pattern-matching threads for one FIXIDS instance.

6.3.3 Vermont Flow Probe

For the experiments in Section 6.4 and in Section 6.5.1 we used Vermont as a Flow probe. In this configuration Vermont captures packets from a NIC or reads them from a stored pcap trace file and then aggregates the packets to IPFIX Flows according to the given configuration.

Among the required IPFIX IEs for intrusion detection with FIXIDS, is the httpRequestTarget IE which contains the HTTP URI. The length of this field is variable and has to be defined in the Vermont configuration. We evaluated the URI lengths of web traffic of a scientific workgroup over a week and came to the following results: the average length of the used URIs in this traffic is 99.8 Byte, the median length 55 Byte, the maximum length 2913 Byte, and the 85 % percentile is 143 Byte. This coincides with the results of a similar survey conducted by Google and published in the SPDY whitepaper[34]. Following these findings we configured Vermont to aggregate the first 150 Byte of the HTTP URI.

It is important to note, that in the configurations where Vermont is exporting HTTP Flows only, if a Flow does not contain HTTP it is not exported. Vermont applies a more sophisticated HTTP detection method by checking for a valid HTTP header. Many other network monitoring tools and Flow probes only offer port-based differentiation of Flows.

For the transport of IPFIX Flows between a Vermont Flow Probe and a Vermont Flow Collector we used the PR-SCTP protocol [150]. In experiments not shown here, this protocol has proofed to be the most effective for this case.

6.3.4 nProbe Flow Probe

It is important to us to show that FIXIDS also works with third-party Flow probes/-Exporters. Because our Cisco Catalyst 4506-E switch has no IPFIX Flow aggregation capabilities, we also evaluated the functionality of FIXIDS taking as input IPFIX Flows from Ntop's Flow Exporter Nprobe (Version 8.6) [144]. Nprobe is a network probe with a broad functionality range available both as software or hardware (branded Nbox).[35] Nprobe is capable of exporting HTTP IEs using the appropriate HTTP plugin. Until now Nprobe uses so called enterprise specific Flow fields to export the information and not yet the IANA standardized HTTP IEs. The content format of the fields, though, is exactly the same. Nevertheless, this required us to extend FIXIDS with a configuration switch to also accept these proprietary fields. For a

[34] `http://dev.chromium.org/spdy/spdy-whitepaper`
[35] `http://www.ntop.org/products/netflow/nbox/`

better comparison with the Vermont Flow probe we limited the Flows to HTTP by configuring Nprobe to only aggregate TCP Flows with destination port 80. This proved to export all HTTP Flows from the traffic that we use in our experiments. Experiments have shown that TCP is the most efficient transport protocol for IPFIX between Nprobe and FIXIDS, thus, we used TCP as the default transport protocol with Nprobe in the experiments below.

6.3.5 Network Setup

We employed one or more of the following workstations in all of the experiments: *Workstation 1* and *Workstation 2* are powered with an Intel Xeon i7-3930K CPU and 32 GByte of RAM. They are both equipped with Intel's 82599 10 Gbit/s dual port NIC. Workstation 1 and Workstation 2 are connected through a Cisco Catalyst 4506-E 10 Gbit/s switch. The switch is necessary to make sure that the routing has no performance impact on the experiment results. Workstation 2 is additionally connected to Workstation 3 with a dedicated 1 Gbit/s link. *Workstation 3* is powered by an Intel i7-7700K CPU and 32 GByte of RAM. All workstations run the Ubuntu Operating System (OS) 16.04 with kernel 4.4.0.

6.3.6 Used Detection Rules

Similar to the other chapters we used the signatures of the following three sources (as of June 22nd, 2018):

- Snort rules (snapshot 29111) provided to Snort.org subscribers.
- Community rule-set from Snort.org.
- All rules from Emerging Threats.[36]

Again, we only used FIXIDS supported HTTP-related rules. In total we applied 5540 rules.

6.3.7 Attack Network Traffic

Choosing the traffic for evaluating a NIDS is a difficult task. For a more in-depth discussion on this topic please refer to Chapter 7, where we address this challenge and present a dedicated malicious traffic generator called GENESIDS. GENESIDS uses HTTP-related Snort rules as attack descriptions and automatically generates one TCP flow with one HTTP request per input rule. The generated HTTP request contains all patterns from this rule and should thus trigger the corresponding event in a NIDS using the same Snort rules. All the rules in the evaluation contain patterns that are

[36] `http://doc.emergingthreats.net/bin/view/Main/AllRulesets`

applied to HTTP requests only. Therefore, GENESIDS only generates HTTP requests. For easy evaluation GENESIDS adds a custom HTTP header containing the unique rule SID.

6.3.8 Realistic Network Traffic

To satisfy the need for realistic benign traffic we employed Cisco's TRex traffic generator (version 2.41) in our experiments.[37] The goal of TRex is stateful traffic generation in a timely precise manner at very high speeds (up to 200 Gbit/s). The advantage of TRex compared to other traffic generators is that it can also include custom generated application-layer payload [151]. For a more in-depth description of TRex and how it can be combined with GENESIDS please refer to Section 7.3.4.

Figure 6.3 shows our evaluation setup. We were using TRex as a traffic generator, the switch (Cisco Catalyst 4506-E) routed the traffic between the two TRex devices and mirrored this network traffic to our FIXIDS system for analysis. We maintained this setup for all of the performance measurements presented in this chapter.

TRex ships with a couple of realistic traffic templates. For the following experiments we used the template defined by the French Telco provider SFR France.[38] It was created to represent typical internet traffic and, according to the TRex manual[39], is used by Cisco to benchmark their ASR1k/ISR-G2 series of routers. The basic composition of this traffic is shown in Table 6.3. Because it only contains benign traffic, it does not trigger any alerts in Snort or FIXIDS with the 5540 rules used in

[37] http://trex-tgn.cisco.com
[38] https://www.sfr.fr
[39] https://trex-tgn.cisco.com/trex/doc/trex_manual.html

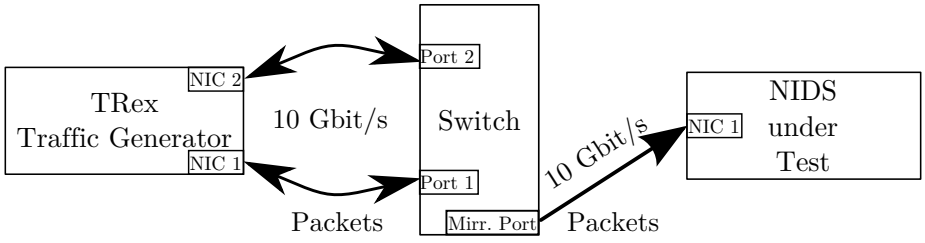


Figure 6.3 – Test setup for the network throughput evaluation experiments.

Table 6.3 – Statistical properties of the SFR network traffic generated by TRex.

Protocol	Packets	Bytes	Connections Per Second (CPS)
TCP	61%	69%	2059
→ HTTP	32%	49%	1519
UDP	39%	31%	2004

this evaluation. For the rest of this chapter we will call the traffic generated with this template "SFR" traffic.

To determine the impact of data losses due to overloading on the detection accuracy we needed a mixed traffic trace, containing benign but also malicious traffic triggering alerts. To make sure to include different traffic properties we generated two mixed traffic traces: We composed the first trace by generating 120 s of the SFR traffic at a rate of 1 Gbit/s resulting in a trace of nearly 25 000 000 packets. We added malicious traffic by adding 500 random attacks generated with GENESIDS using 500 random Snort signatures from the set of the above 5540 rules. We distributed the attacks evenly over the whole trace. Then we concatenated this trace six times. We rewrote the IP addresses in a consistent way, making sure the addresses changed in all of the 6 replications but keeping single TCP flows. With Vermont in the Flow probe configuration as used in the experiments, 1 485 000 Flows are exported. For the rest of this chapter we will call this trace "SFR+500x6" trace.

We composed the second trace by taking the traffic of a scientific work group going through an HTTP proxy for one week. To add more malicious traffic we mixed it with 500 attacks as with the SFR+500x6 trace. Again, we concatenated the trace 35 times anonymizing IP addresses as above. Vermont as Flow probe exports 1 494 000 Flows for this trace. From now on this trace is called "PROXY+500x35 trace".

6.4 Functional Evaluation

In this section we present experiments to evaluate the functionality of FIXIDS. We focus on the accuracy of the detection method by using a broad variety of malicious traffic generated with GENESIDS. The attack descriptions used as input for GENESIDS and FIXIDS are the ones described in Section 6.3.6. The setup for the functional evaluation experiments is sketched in Figure 6.4. The experiment consisted of two steps: In step one we created the attack traffic and determined the baseline of malicious events to be found. In step two we analyzed the traffic with FIXIDS and compared the detection results with the results of Snort.

From a technical perspective this works the following way: *Step 1:* GENESIDS ran on Workstation 1. The input were the 5540 rules determined in Section 6.3.6. GENESIDS created an HTTP request for each of the attacks and sent it to Workstation 2. On Workstation 2 an Apache webserver (version 2.4.10) answered these requests. Additionally, we captured all the traffic with the network traffic capturing tool `tcpdump`[40] and saved it to a pcap network trace. *Step 2:* Both NIDS under test, Snort and FIXIDS were configured to analyze the pcap network trace captured in

[40] `tcpdump.org`

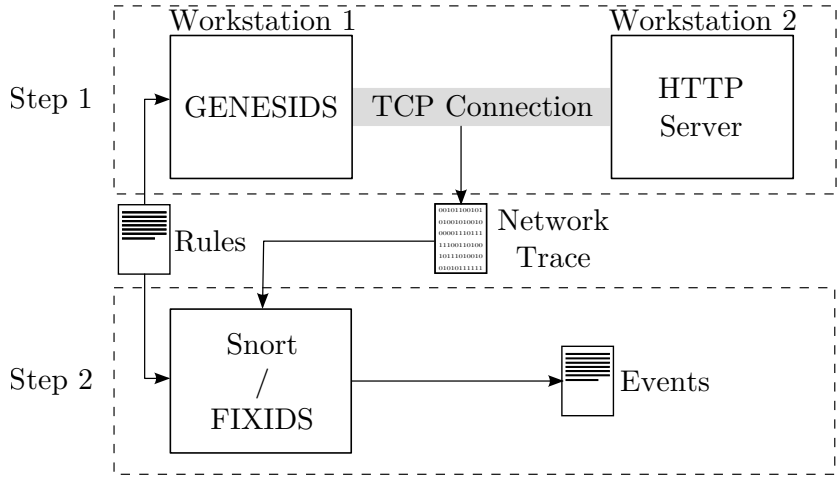


Figure 6.4 – Sketch of the functional evaluation experiments.

Step 1. Both took as input the same rule file containing 5540 rules that was used by GENESIDS to generate this traffic and both NIDS wrote the detected alerts to a separate event file. To be able to use the pcap trace with FIXIDS we needed to convert it to IPFIX Flows first. Therefore we configured Vermont to not only do intrusion detection with the FIXIDS module but also to act as a Flow probe. This way the packets of the pcap trace were aggregated to IPFIX Flows before handing them to the FIXIDS module.

Finally, we compared the event files of Snort and FIXIDS. The events reported by Snort represent the benchmark that we compared FIXIDS to. Here we want to emphasize that we did not simply check if Snort and FIXIDS triggered any event for a malicious HTTP request, but made sure that the triggered event corresponds exactly to the attack that this HTTP request was written for. This is what we denote as a "true positive". If the triggered event does not correspond to the attack in the HTTP request we tag it as a "false positive". The mapping between triggered events and generated attacks in the HTTP request is made by comparing the TCP source port and the SID number reported in the event file, with the TCP source port and the SID number used by GENESIDS during traffic generation. This requires that GENESIDS uses a unique port number for every issued request, which is the default case but was verified manually beforehand.

We took advantage of the nondeterministic traffic generation of GENESIDS to examine the detection robustness of FIXIDS, by repeating the experiment a hundred times. The nondeterminism of GENESIDS is firstly caused by certain Regular Expression (RegEx) patterns which produce a different traffic in every run, and secondly by the fact that we changed the order of the input rules and, thus, also the order of the generated HTTP attack requests changes with every run.

The results of this experiment are shown in Figure 6.5. Firstly, we can verify that GENESIDS generated exactly 5540 (100 %) attack requests in every experiment run. Secondly, the plot shows that Snort and FIXIDS have an analogous, consistently high true-positive detection rate of more than 99 %. On average, Snort detects 5489 true positives of the generated attacks (max. 5494 and min. 5481) and FIXIDS marginally more with 5490 true positives (max. 5495 and min. 5483). Considering that we used 5540 different attack descriptions and that FIXIDS shows the same very high detection accuracy than the state-of-the-art NIDS Snort, we conclude that FIXIDS has passed this evaluation step with honors.

The reason that both NIDS did not detect all of the generated events is because GENESIDS, in very rare cases, is not able to generate the HTTP request according to the input attack description (Section 7.4 gives a detailed explanation of this behavior). In such cases, it is impossible for the NIDS to detect the correct event. Please mind that the experiment includes an unprecedented high amount of different attacks, and this behavior is only the case for less than 1 % of the attacks and, thus, negligible for our purposes.

The false-positive events detected by both NIDS during the 100 runs of the experiment are shown in Figure 6.6. Snort and FIXIDS both triggered a relatively high amount of an average of 2530 and 2519 false-positive events respectively. This is not unusual if the applied rule-set contains such a high number of rules. The reason behind this is the overlap of rule patterns as described by Massicotte and Labiche [152]. This causes multiple rules to trigger an event for the same packet. In a real environment, the applied rules are chosen very carefully and adapted to the application scenario. For example, in the above experiment 74 % of the false positives could have been avoided by disabling only three rules.

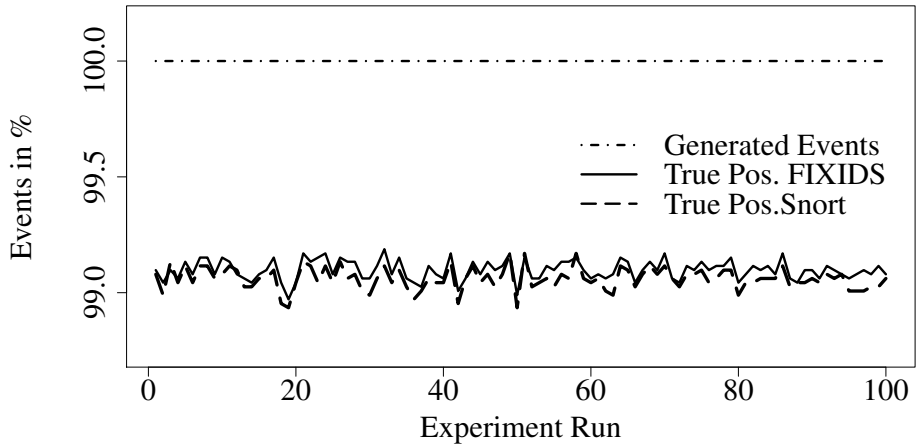


Figure 6.5 – Detected true-positive events by Snort and by FIXIDS in 100 different attack traces generated with GENESIDS.

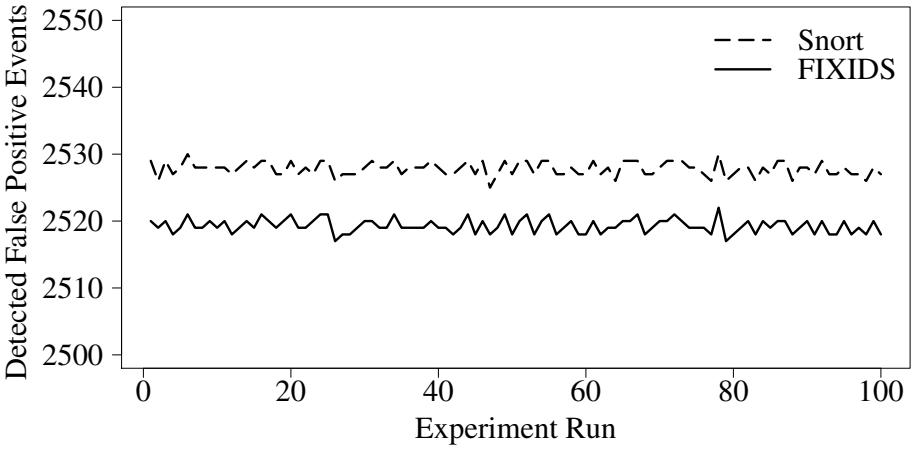


Figure 6.6 – Detected false-positive events by Snort and by FIXIDS in 100 different attack traces generated with GENESIDS.

In the last step of our experiment analysis we compare the differences of true-positive events detected by both NIDS. While the discrepancy between the two systems is negligible (maximum of 11 events per run), the investigation of these events helps to better understand the details of the single detection engines. We start with the true-positive events detected by Snort and missed by FIXIDS:

In total Snort detected 624 true positives that FIXIDS did not detect. In the vast majority of the cases (more than 83 %), the reason was that the data part containing the relevant pattern was located in the URI beyond the 150 Byte limit captured by Vermont and, thus, invisible for FIXIDS (cf. Section 2.4). This can be avoided by configuring the Flow probe to retain more data of the HTTP URI. For the remaining events we identified the following causes:

- One of the generated HTTP requests contains “post” as the request method. Because the standard defines HTTP methods as case sensitive, Vermont does not recognize this HTTP request as standard conform and, thus, dismisses it during the IPFIX aggregation process which leads to FIXIDS never analyzing this request. This attack was missed once every run (100 times in total).

- In one experiment run, GENESIDS generated an HTTP request including a double slash in the HTTP URI. In the Snort default configuration that we use for the experiments, the *http_inspect* preprocessor normalizes this and removes double slashes at all, which caused this URI to still match the corresponding attack pattern. FIXIDS analyzes the URI without removing the double slashes and, thus, does not trigger the corresponding event.

Finally we investigate the true-positive events detected by FIXIDS and missed by Snort: This involves 834 events over 100 runs generated by 65 different rules (min

six per run, max 11). Most of the missed true positives are caused by one of the Snort preprocessors normalizing the data before intrusion detection analysis:

- If an HTTP request contains a "\\", it is converted to "//" (in the default configuration). This entails that the request will not match the corresponding pattern anymore.
- If the URI in an HTTP request starts with a "/" followed by a "+", the "+" character is removed by the *http_inspect* preprocessor. The reason for this is that Snort interprets the "+" as a whitespace which is not expected at this position, thus, it removes it before analysis.
- Some generated HTTP requests contain a URI starting with "http\ :" which is normalized to "http://". Ergo, the corresponding rule does not match anymore.
- The rest of the missed true positives by Snort occurred less than 5 out of 100 runs. They were missed mostly because of different interpretations of special characters (mostly whitespace characters) by Snort and FIXIDS.

We conclude the evaluation of the detection accuracy by stating that FIXIDS has the same, very high detection rate (>99 %), as Snort. This was determined in an extensive experiment campaign including more than 5500 different attacks. This underlines that the detection functionality of FIXIDS is precise and reliable.

6.5 Throughput Performance Evaluation

The experiments presented in this section assessed the network throughput performance of FIXIDS under different scenarios. Firstly, we evaluated the baseline throughput performance of FIXIDS. Then we assessed the network throughput capability when fed with IPFIX Flows from a third-party Flow probe and, finally, we conducted an experiment, mimicking a realistic scenario, where FIXIDS is used to remove the load of a legacy signature-based NIDS. The metric that we use to express the throughput performance throughout the experiments is IPFIX Flows/s. In scenarios where we compare the throughput to Snort we also convert this to Gbit/s.

6.5.1 Basic Throughput Experiments

With the following experiment we assessed how much Flows/s a single instance of FIXIDS can handle without dropping any data. To evaluate the impact of different traffic properties we used the two realistic network traffic traces SFR+500x6 and PROXY+500x35 presented in Section 6.3.8.

As we are only interested in the traffic throughput of FIXIDS, we excluded possible latency sources like routing appliances and Flow probes by generating the Flows in

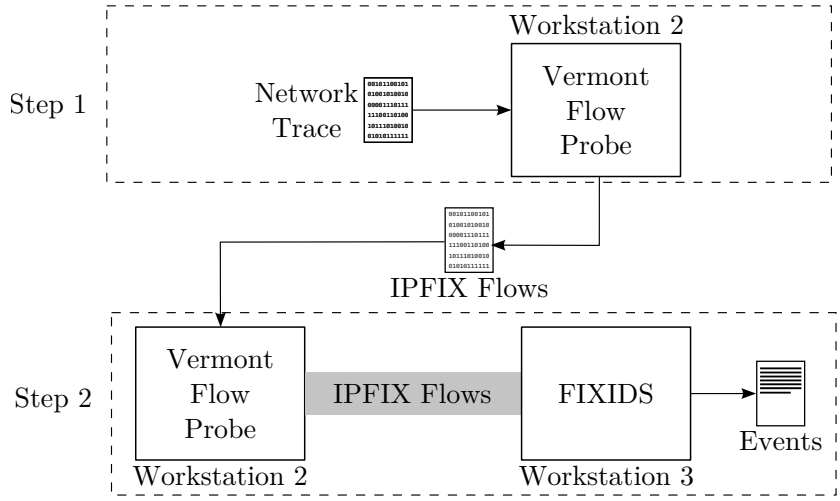


Figure 6.7 – Single steps of the basic throughput experiments.

a preparation step. The single experiment steps are sketched in Figure 6.7. *Step 1:* A Vermont Flow probe took the network trace as input and aggregated it to IPFIX, including the necessary HTTP IE fields. Please mind, we aggregated and exported HTTP traffic only. The resulting Flows were then written to a Vermont specific binary file. *Step 2:* A Vermont probe on Workstation 2 took the readily prepared IPFIX Flows from the binary file and sent them to a FIXIDS instance on Workstation 3. The FIXIDS instance performed intrusion detection applying the 5540 rules, using the same configuration as used in the experiments in Section 6.4 and depicted in Figure 6.1. We repeated Step 2, steadily increasing the replay rate in 1000 Flows/s steps from 1000 Flows/s to 30 000 Flows/s.

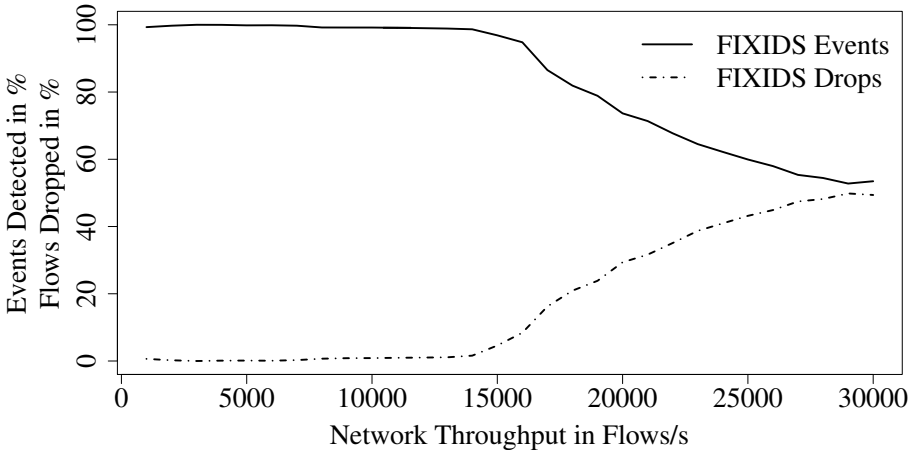


Figure 6.8 – Flow throughput performance of FIXIDS analyzing the SFR+500x6 trace (average of ten runs).

The results are plotted in Figure 6.8 for the SFR+500x6 trace and in Figure 6.9 for the PROXY+500x35 trace. The results are the average of ten runs. For readability reasons the confidence intervals (95 %) are only shown if above 2 %.

Looking at the results with the SFR+500x6 trace, FIXIDS starts dropping packets only at 14 000 Flows/s. At the same throughput rate, also the detected event ratio drops because the events contained in dropped Flows can not be detected anymore. The throughput rate with the PROXY+500x35 trace is with 11 000 Flows/s slightly lower. Also the ratio of detected events drops in a much more dramatic way. This is because in this trace a small number of rather large Flows contain many events, including a high number of false positives. These Flows stretch over multiple packets and, thus, the probability that a packet belonging to these Flows gets dropped is comparably high. If a packet is missing FIXIDS is not able to reconstruct the Flow. This entails that larger Flows suffer more from packet drops than small Flows.

The reason for the lower throughput rate with the PROXY+500x35 trace is the following: There are considerably more HTTP GET requests contained in the PROXY+500x35 trace compared to the SFR+500x6 trace. This has a negative impact on the performance, because most rules apply a pattern looking for a “GET” in the HTTP method. If this pattern is not found, which is much more likely with the SFR+500x6 trace, the pattern-matching thread will drop this Flow and continue with the next one. With the PROXY+500x35 trace, the “GET” is found more often and then the next pattern is applied (in most cases a complex RegEx) to the same Flow, which means that the pattern-matching step will, on average, spend more time with a single Flow compared to the SFR+500x6 trace.

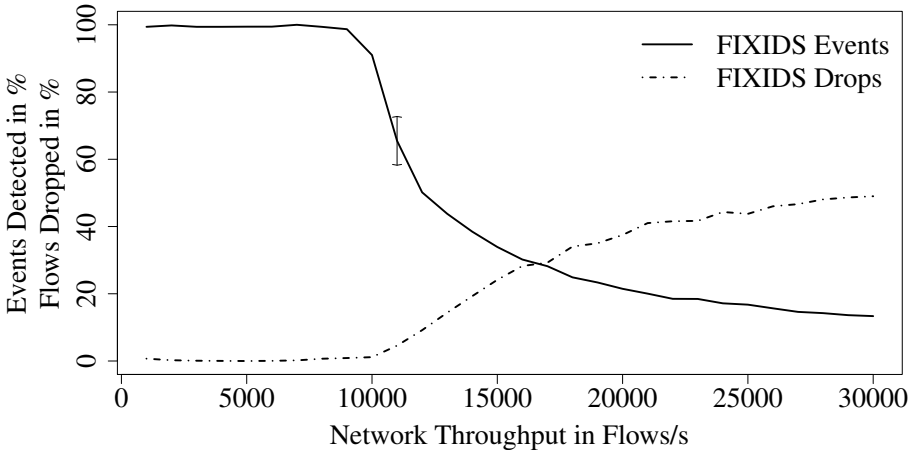


Figure 6.9 – Flow throughput performance of FIXIDS analyzing the PROXY+500x35 trace (average of ten runs).

6.5.2 Third-Party Flow Exporter Experiments

With the following experiment we evaluated the FIXIDS throughput performance if Flows are aggregated and provided by a third-party appliance. We consider such an interoperability scenario necessary because this is what will typically happen in real scenarios, where network monitoring appliances from different manufacturers are combined. The realistic traffic was generated with TRex on Workstation 1 as depicted in Figure 6.3. The traffic consists of 5 min of SFR traffic mixed with 100 random attacks from the set of attacks used in Section 6.4. The traffic was routed through the switch and mirrored to the NIDS under test. The used NIDS under test were FIXIDS (in combination with Nprobe) as depicted in Figure 6.10 and, for comparison reasons, Snort (see Figure 6.11).

In the former case, the Nprobe Flow Exporter on Workstation 2 received the network traffic generated by TRex from the mirroring port of the switch. Because of the high traffic rates we combined 6 Nprobe instances for this task, by taking advantage of its clustering capability (using the *–cluster-id* switch). This equally distributed the network traffic, according to its IP addresses and ports, to all instances of the Nprobe cluster. Therefore, the performance intensive task of aggregating packets to IPFIX was distributed among multiple instances and a higher packet throughput rate can be achieved.

The IPFIX Flows of Nprobe, including the necessary HTTP IEs, were then sent via a 1 Gbit/s link to FIXIDS running on Workstation 3. We want to emphasize that a line rate of 1 Gbit/s is more than sufficient for the transport of IPFIX Flows in such a scenario. As a matter of fact, we never fully used the bandwidth in any of our experiments. Also here we combined two instances of FIXIDS to distribute the intrusion detection analysis load. This was done by instructing three Nprobe

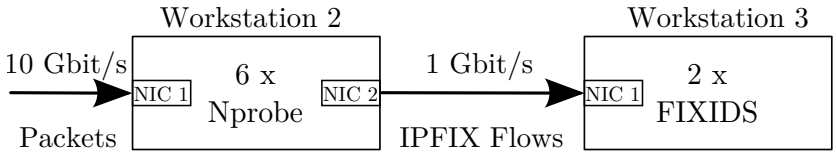


Figure 6.10 – NIDS under test: 6 Nprobe instances aggregate the incoming packets to IPFIX Flows and send the Flows to 2 FIXIDS instances.

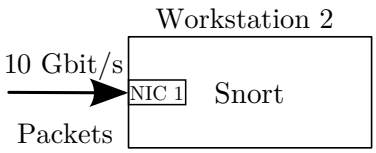


Figure 6.11 – NIDS under test: Snort directly analyzes the incoming packets from the switch.

instances to send their Flows to the port of the first FIXIDS instance, and the other three Nprobe instances to use the port of the second FIXIDS instance. Again, we applied all 5540 HTTP rules for Snort and FIXIDS. Because we were interested only in HTTP traffic, and for a better comparison with the results from Section 6.5 we filtered the traffic (by HTTP ports) in front of Nprobe and Snort.

We continuously increased the CPS multiplier of TRex in steps of 0.5 Gbit/s, resulting in a throughput rate of 0.5 Gbit/s up to 9.5 Gbit/s. It is important to note that the throughput rate was measured at the TRex traffic generator for the full traffic, but both NIDS under test only analyzed the filtered HTTP traffic part.

This experiment setup manifests once more the advantages of Flow-based intrusion detection. Firstly, the data aggregation step can be done on an own, dedicated device, and secondly, because standard transport protocols are used, the distribution of the traffic to multiple intrusion-detection engines can be accomplished simply by using different destination ports. This is both not possible in such a straight-forward way with legacy, packet-based NIDS where these steps happen in a tightly coupled fashion.

The averaged results over ten runs are plotted in Figure 6.12. The confidence intervals (95 %) are only shown if above 2 %. The plots show the events detected by Snort and by FIXIDS as well as the dropped packets and Flows over different throughput speeds.

Snort is able to cope with a rate of up to 2 Gbit/s, after that it starts to randomly drop packets and, thus, also the event detection ratio decreases. FIXIDS can cope with a significantly higher rate, showing no Flow drops or decreases in the detection ratio even at the highest rate of 9.5 Gbit/s. This is more than four times faster than

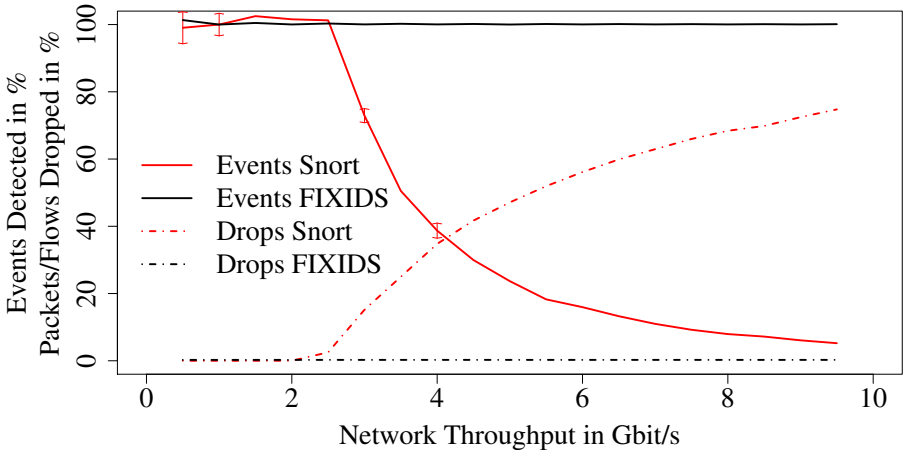


Figure 6.12 – Event detection rate of Snort and FIXIDS analyzing the packets / IPFIX Flows of 5 min of SFR traffic including 100 events per second (average of ten runs).

Snort. The detailed statistics of a single FIXIDS instance reveal that the maximum Flow rate was 11 000 Flows/s and, thus, much lower than the maximum rates with SFR traffic for FIXIDS found out in the previous Section 6.5.1.

6.5.3 Real World Scenario

In this section we present an experiment that imitated as close as possible how FIXIDS can be deployed in an existing IT security scenario, as depicted in Figure 6.13. Before the deployment of FIXIDS the bypassing network traffic was mirrored by the switch to Snort which analyzed it for intrusions. We then configured a second mirroring port and filtered the traffic, so that FIXIDS received HTTP traffic only and Snort received the remaining non-HTTP traffic. This allowed that FIXIDS could apply all HTTP-related rules and Snort the remaining non-HTTP rules. This way the load was distributed between FIXIDS and Snort.

The experiment setup was the following: We measured the baseline performance in a first experiment step, where Snort analyzed the complete traffic and applied all rules. In this step, the rule-set did not only contain the previously used 5540 HTTP rules supported by FIXIDS, but additionally 5714 non-HTTP-related rules. This totaled to a rule-set of 11254 rules. TRex generated traffic using the SFR traffic template (cf. Section 6.3.8). We have tested the attack detection performance of FIXIDS sufficiently so we did not add any attack traffic. The network throughput was increased continuously by manipulating the CPS multiplier of TRex. We started with a throughput of 0.5 Gbit/s and increased it until 9.5 Gbit/s in 0.5 Gbit/s steps.

In a second experiment step we included FIXIDS in this scenario. First, we configured a second mirroring port at the switch so that FIXIDS received the same

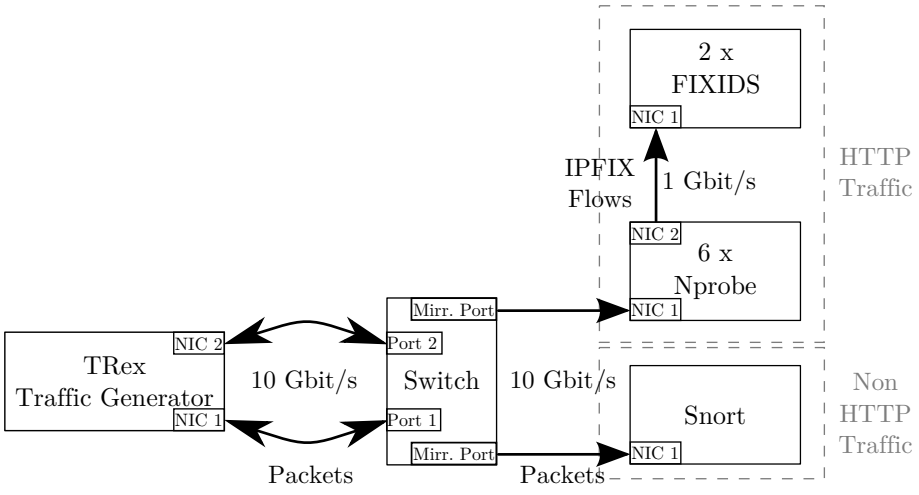


Figure 6.13 – Real world scenario; FIXIDS is used to reduce the load of Snort by taking over the HTTP part of the traffic and the HTTP-related rules.

traffic as Snort. Then we filtered the traffic at both mirroring ports with the help of TCP port numbers. Snort, in its default configuration, defines a number of HTTP ports. We used this list of HTTP ports to forward all HTTP traffic to FIXIDS and all non-HTTP traffic to Snort. Then we configured Snort to apply all 5714 non-HTTP rules and FIXIDS to apply all 5540 HTTP rules. The Snort and the FIXIDS configurations were the same as in Section 6.5.2. This means that we used six Nprobe instances on Workstation 2 to aggregate IPFIX Flows and export them to two FIXIDS instances on Workstation 3.

Figure 6.14 shows the average of 10 experiment results. No confidence intervals (95 %) are shown because they are always below 2 %. The baseline performance of Snort, analyzing all traffic and applying all rules, is shown with the solid line. It reveals that Snort, in this scenario, can cope with about 0.5 Gbit/s of network traffic without packet drops. The smaller network throughput compared to earlier experiments is owed to the higher number of rules, which now more than doubled.

As soon as we included FIXIDS in the scenario, and removed traffic and rules from Snort, the network throughput without drops triples to 1.5 Gbit/s. Theoretically, we would have to add also the drop rate of FIXIDS, which applies the remaining HTTP rules to the HTTP traffic. However, both FIXIDS and Nprobe did never drop any packets, not even at the maximum rate of 9.5 Gbit/s. A closer look at the Flow rates shows that FIXIDS had to handle a maximum of 8400 Flows/s per instance which is considerably lower than the maximum Flow rate assessed in earlier experiments for this kind of traffic. Thus, FIXIDS can not only remove a considerable amount of load from an existing intrusion detection system, but, in our proposed experiment scenario, it could very likely apply even more rules without having to drop any packets.

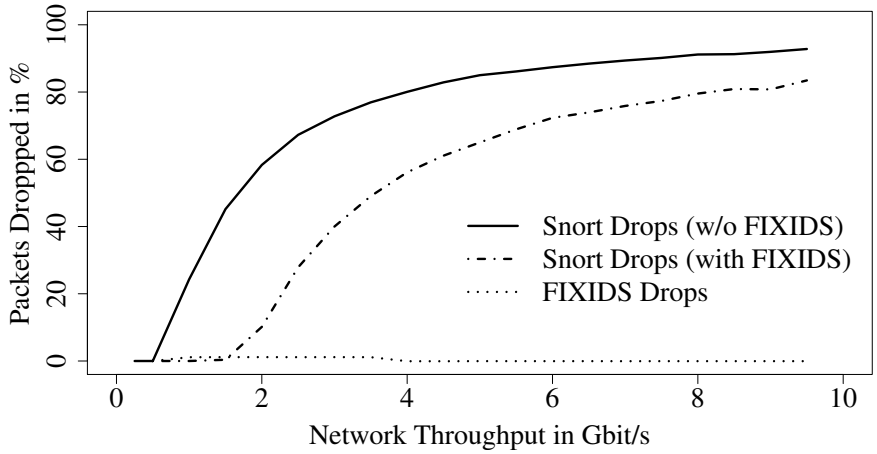


Figure 6.14 – Packet drop rates of the realistic scenario were FIXIDS is deployed to reduce the load of Snort (average of ten runs).

6.6 Lessons Learned

In this chapter we presented our novel, high-speed NIDS FIXIDS. FIXIDS can cope with higher network throughput speeds than legacy NIDS by using HTTP IPFIX Flows for intrusion detection. This considerably reduces the amount of data to be analyzed compared to legacy DPI-based NIDS. To the best of our knowledge, this is the first signature-based NIDS analyzing IPFIX Flows and taking advantage of payload-based IEs. By using Snort signatures we guarantee that thousands of community validated and up-to-date signatures are available, while still allowing to define own attack descriptions for new or application-scenario-specific attacks.

The evaluation experiment results show that FIXIDS has the same, very high detection accuracy as Snort. Using commodity hardware, a single FIXIDS instance can cope with up to 14 000 Flows/s without dropping any data or missing any intrusion events. When analyzing the same network traffic, FIXIDS can cope with more than four times the network throughput of Snort, while retaining the same detection rate, detecting HTTP-based intrusions even at 9.5 Gbit/s without drops. In a realistic intrusion detection scenario experiment, FIXIDS is able to remove a considerable amount of load from Snort. In this scenario the network throughput capability of Snort could be tripled with the help of FIXIDS. Hereby, we successfully answered research question three: Can we use HTTP-enriched IPFIX Flows for efficient intrusion detection.

Chapter 7

GENESIDS: An Automated System for Generating Attack Traffic

In the previous chapter we presented our novel IPFIX-based Signature-based Intrusion Detection System (FIXIDS). In this chapter we propose a malicious Hypertext Transfer Protocol (HTTP) traffic generator, which, with the help of Snort attack signatures, is able to generate an unprecedented variety of different attacks.

This chapter is based on the following publications:
F. Erlacher and F. Dressler, "How to Test an IDS? GENESIDS: An Automated System for Generating Attack Traffic," in *ACM SIGCOMM 2018, Workshop on Traffic Measurements for Cybersecurity (WTMC 2018)*, Budapest, Hungary: ACM, Aug. 2018, pp. 46–51
F. Erlacher and F. Dressler, "Testing IDS using GENESIDS: Realistic Mixed Traffic Generation for IDS Evaluation," in *ACM SIGCOMM 2018, Demo Session*, Budapest, Hungary: ACM, Aug. 2018, pp. 153–155

7.1 Motivation

All Network Intrusion Detection Systems (NIDS) developers face one common task: they have to validate the developed system and make sure that it works as expected in all possible scenarios. Multiple publications [153], [154] propose and summarize methods on how to evaluate a NIDS, and they all agree that one of the major steps is to test the attack coverage and detection precision. In this chapter we propose a framework that is going to simplify this task significantly.

A straightforward method to test the detection abilities of a NIDS, is to use traffic from a live network or publicly available traces [155], [156]; in the best case with some annotations about the ground truth of the contained malicious traffic. Usually public network traces are used, as only very few developers and researches have access to networks big enough to represent realistic traffic. But such traces come with a couple of drawbacks [157]: because of privacy reasons most of these traces do not contain any application-layer payload and most of them are already a couple of years old. However, the biggest challenge with such traces is that they contain a low number of attacks in total and only a fraction of all attacks that a NIDS has to detect in reality. Developers and researches usually face this challenge by manually creating additional malicious traffic and then multiply and distribute this traffic over an existing trace, as in Erlacher and Dressler [26], Bul'ajoul et al. [158], and Lukaseder et al. [159]. This is also what we did in earlier work (cf. Chapter 4, Chapter 5). But manually creating attack traffic is very cumbersome and, thus, the low number of added manual attacks can be compared to the proverbial drop in the ocean. Therefore, and this has been frequently pointed out by reviewers, such traces do not convince that the attack coverage evaluation is comprehensive.

To solve this challenge we propose the GENESIDS (Generating Events for Signature-based Intrusion Detection Systems) traffic generator. It takes as input a set of user defined attack descriptions and then statefully generates network traffic including the patterns of the given attacks. The NIDS under test should then be able to detect these attacks, or a subset thereof, depending on the application scenario. The main advantage of our system, is that it uses the Snort signature syntax as attack description language and, thus, directly accepts many signatures for the Snort NIDS. As a result, the time consumption of manually defining attacks is reduced to a minimum because Snort provides thousands of up-to-date and community validated signatures (also called rules) assuring a broad coverage of real-world attacks.

Because Snort signatures are broken up in different categories it is easy to choose the proper signatures for the specific NIDS under test. Nevertheless, if the application scenario requires to generate attacks not known to the Snort signature databases, they can be created by manually defining a corresponding attack description. Additionally, we facilitate the evaluation of the attack coverage by labeling every created attack packet with its unique rule SID number. Within the set of Snort signatures we focus on rules that are applied to the HTTP protocol. It is the most used application-layer protocol [140] and covers most events detected by a NIDS (cf. Section 5.2).

Until now, the created traffic of GENESIDS contains attack traffic only. To mix this malicious traffic with realistic benign traffic, we propose to use a L4-L7 traffic generator and combine both traffic sources (cf Section 7.3.4). Because such traffic is artificially created, no privacy is harmed and, thus, traces can be published and shared with the community, which allows for a better comparison among different proposed NIDS and increases the repeatability of the evaluation of scientific research.

GENESIDS is written in C++ and is publicly available under a GPL license.[41]

7.2 Related Work in Traffic Generation

Kernel bypass network APIs like Intel's DPDK[42] or Ntop's PF_Ring [146] have paved the way for affordable, precise and reliable traffic generators following given time or burst patterns at very high data rates [160]. Their main application scenarios are load tests and other high-speed performance evaluations. Only few of them are also able to create user-definable application-layer payload. And to the best of our knowledge, none of the currently available traffic generators can automatically generate attack traffic including real-world malicious payload.

Typically, publications [161]–[163] of novel NIDS use public network traces for their evaluation. Apart from the already mentioned old age, these traces are heavily criticized [164] because of their debatable accuracy, timeliness, and completeness.

[41] github.com/felixe/idsEventGenerator
[42] `http://www.dpdk.org`

This does not necessarily hold for the captured traffic of hacking contests or cyber-warfare exercises [165]. Unquestionably, they will contain more attacks than the criticized public network traces, but the contained malicious traffic will not be very diverse, because the attack task of such campaigns usually include only a low number of target systems and applications.

Manually creating attack traffic can be facilitated by using penetration testing frameworks like Metasploit[43] as in Lukaseder et al. [159] and Nasr et al. [166]. While this eases some single steps in the crafting process, it is not enough to create a sufficiently high number of different attacks.

Summarizing, we conclude that today's traffic generators show problems similar to public traces: the generated traffic does not contain enough *unique* attacks to comprehensively evaluate the attack coverage of NIDS. As we will see in the next sections, GENESIDS not only allows to create handcrafted attacks, but, by using Snort rules, already includes more than 8000 up-to-date HTTP attacks. The problem of realistic timings has not been tackled with GENESIDS. We propose to use traffic generators like TRex[44] or moongen [167], which are capable of delivering precise timing on commodity hardware, in combination with GENESIDS (cf. Section 7.3.4).

Attack traffic generation frameworks, including GENESIDS, can also be used for so called squealing attacks [168]. With this kind of attack, malicious traffic is used to trigger millions of alerts in a short time, to overload the attacked system and the network operator with alarms. Fortunately, modern NIDS and firewalls have evolved from simple pattern checkers to stateful appliances incorporating multiple methods to detect and avoid such evasion techniques.

7.3 GENESIDS Architecture

GENESIDS creates malicious HTTP traffic for use in signature-based NIDS evaluation. Because the Snort database contains the largest set of machine-readable attack descriptions, we decided to accept the Snort signature syntax as input format. As described in Section 1.1, signature-based NIDS detect malicious activity by searching the patterns that are defined in the rules in the bypassing traffic. GENESIDS creates malicious traffic by creating one HTTP packet per rule, containing the exact patterns of this rule. If a signature-based NIDS receives this HTTP request and the same rule is part of its rule-set, it will trigger the event for this rule.

[43] `www.metasploit.com`
[44] `https://trex-tgn.cisco.com`

7.3.1 Input and Connection Management

The first step of GENESIDS is to parse the rules from the given input file. If a rule does not correspond to the syntax, an alert message will be issued. A warning message will be issued if potential problems are detected within a rule. Then, GENESIDS will go through all parsed rules and generate an HTTP request containing all patterns of the corresponding rule. During this step, GENESIDS makes sure that a complete and valid TCP session is created: At startup, the user has to define the address of an HTTP server, this address is used now to send the generated HTTP requests to. GENESIDS expects the HTTP server to respond to the incoming requests. This entails that responses are not controlled by GENESIDS. As soon as the HTTP response arrives (or after a timeout), GENESIDS will close the TCP connection and start the same process again for the next rule. This means that GENESIDS uses a stateful approach: all patterns of a single rule are included in one HTTP request which is again, together with the corresponding HTTP response, included in a single dedicated TCP connection. The reason for this is that most NIDS only consider complete and correct TCP connections

With the used rule-syntax one can determine the following HTTP fields in the HTTP requests generated by GENESIDS: method, Uniform Resource Identifier (URI), header names and values, request cookies, and the client body. The patterns for those fields can be given as ASCII text (including hexadecimal representation of single characters) or using a Perl Compatible Regular Expression (PCRE) (cf. Section 7.3.2).

The HTTP requests are created using the well known `libcurl` library. For every rule, the given content patterns are then copied in the corresponding field of the generated request. If the pattern contains characters in hexadecimal representation they are replaced with the corresponding ASCII character.

If the pattern is defined using a PCRE, the generation is a bit more complex: GENESIDS has to create a matching string for the given PCRE. For this, we use the Python command `exrex`[45], which is also available as a Linux shell command. The advantage of exrex, compared to other tools, is that it has only very few limitations concerning the input Regular Expression (RegEx). The most important limitations are the following: it only supports 7-bit hex chars, it does not support some combinations of quantifiers, and it does not support positive look-ahead. This has proved to cause problems in only very few cases. Nevertheless, to alleviate problems we try to exchange the most common cases of unsupported combinations of quantifiers with equivalent ones. For example, for the generation of traffic the (unsupported) PCRE `/a+?/` is equivalent to `/a+/`, thus, GENESIDS replaces them accordingly.

Another problem that arose is that some NIDS are sensitive to unusual characters, thus, GENESIDS replaces frequently used PCRE parts that possibly create

[45] `pypi.python.org/pypi/exrex`

unusual characters with equivalent but safe ones. An example is the frequent pattern `SELECT.*FROM` which is replaced with `SELECT[a-z]FROM` before using it to create a string with the exrex command. Nevertheless, if GENESIDS should be used to assess the robustness against unusual characters of a NIDS, these characters can still be used by explicitly stating them in the PCRE or in a content pattern. For the rare cases where the string generated from a PCRE still contains problematic characters (e.g., if a reserved character according to Berners-Lee et al. [169] is used in the HTTP URI), GENESIDS will issue a warning. Please keep in mind, that for many RegExes the number of possible matching strings is infinite.

7.3.2 Rules

GENESIDS supports the rule syntax used for HTTP-related signatures as of Snort 2.9.11 and described in the Snort manual[46]. GENESIDS only supports rules with the action keyword *alert*. They are the only rules of interest for malicious traffic generation. Only HTTP-related rules are supported and the only supported transport protocol is TCP. What follows is a description of the accepted fields and keywords:

The accepted rules consist of a rule header followed by the rule options. The first keyword in the rule header is the rule action, the only accepted rule action is “alert”. The rule action is followed by the source and destination address/port pair. This concludes the rule header and an opening parenthesis indicates the start of the rule options. The rule options are composed by a number of keywords possibly followed by a value. During parsing of the rule file GENESIDS checks for unsupported rule options and issues a warning, ignoring the corresponding rule, if possible.

The following list shows the mandatory key-value pairs in the rule options (in order of appearance):

- `msg`: The value of this keyword is the message description of the triggered event.

At least one of the following key-value pairs must be included in a rule:

- `content`: This value consists of the quoted pattern to insert in the HTTP request. For GENESIDS it must be followed by one of the supported HTTP content modifiers (see complete list below).
- `uricontent`: This value consist of a quoted pattern to insert in the HTTP URI of the request.
- `pcre`: This value consists of a PCRE describing the pattern to insert, enclosed in slashes. For GENESIDS it must be followed by one of the supported HTTP PCRE content modifiers (see complete list below).

[46] `snort.org/documents`

For the `content:` key-value pair, an HTTP-related content modifier is required for the rule to be accepted. The following are supported by GENESIDS:

- `http_method` - A pattern followed by this content modifier will be inserted as the HTTP method. The method field is mandatory for a valid HTTP request. Therefore, if no method is given in a rule, GENESIDS defaults to "GET".
- `http[_raw]_uri` - The pattern will be inserted in the HTTP URI. The default value is "/".
- `http[_raw]_header` - The pattern will be inserted as an own header field. If the pattern does not contain a name:value pair, it is assumed that the pattern represents a value and a default key name is inserted.
- `http[_raw]_cookie` - The pattern will be inserted in the HTTP request.
- `http_client_body` - The pattern will be inserted in the HTTP request body. Please note that some rules issue a GET request with an HTTP body content. This is supported by GENESIDS and perfectly legal. But the HTTP standard requires the server to ignore any request body content if not required by the semantic of the method (e.g., in GET requests).

For the `pcre:` key-value pair, an HTTP-related content modifier is required for the rule to be accepted. The following are supported by GENESIDS:

- `M` - Same semantic as http_method above.
- `U, I` - Same as http_uri.
- `H, D` - Same as http_header.
- `C, K` - Same as http_cookie.
- `P` - Same as http_body.

The Snort HTTP content modifiers applied to HTTP responses (`http_stat_msg` and `http_stat_code` and `Y` and `S` for `pcre:` patterns) are not supported because GENESIDS only generates HTTP requests and has not control over the HTTP responses.

The other two mandatory keywords required by GENESIDS are `sid:` and `rev:`. The first one is used to assign a unique identifier to this rule, and the second one indicates the revision number. They are both followed by an integer number.

If other keywords (e.g., `flow:from_server` or `to_client`) are detected that indicate that the rule should be applied to an HTTP response, GENESIDS will issue a warning and ignore the rule.

To include the attack ground truth in the traffic and, thus, ease the evaluation of experiments conducted with GENESIDS, we add the unique SID number of the

rule associated to the HTTP request as an additional header field (as `Rulesid:<sid number>`). This eases a scripted false / true positive evaluation of the triggered events. An accepted rule could look like the following:

```
alert tcp any any -> any any (msg:"Example rule";
content:"GET"; http_method;
uricontent:"|2F|mallory.zip";
sid:012345; rev:42;)
```

The HTTP request generated by this rule will contain a “GET” as the HTTP method and the string “/mallory.jpg” as the HTTP URI (note the conversion from the hex byte ’2F’ to its ASCII representation ’/’). Additionally, the SID is added as HTTP header field. A textual representation of the generated request could look like the following (assuming the HTTP server address is configured to be 10.0.0.1):

```
GET /mallory.zip HTTP/1.1
Host: 10.0.0.1
Rulesid: 012345
```

The final TCP flow will also include the HTTP response from the server as well as the TCP handshake and teardown.

An accepted rule using a `pcre:` encoded pattern could look like the following:

```
alert tcp any any -> any any (msg:"Example pcre rule";
content:"POST"; http_method;
pcre:"/EvilBody[0-9].*/P";
sid:012346; rev:0;)
```

The `pcre:` pattern will generate a string starting with “EvilBody” followed by a single digit, followed by a random string. Please note, to avoid possible irritating characters the `.*` part of the PCRE will be replaced by the equivalent (for traffic generation purposes) expression `[a-z]`. As stated in the list above, the content modifier `P` indicates that this pattern is inserted in the HTTP request body. Analogous to the previous rule, the textual representation of the generated HTTP request will look like the following:

```
POST / HTTP/1.1
Host: 10.0.0.1
Rulesid: 012346
Content-Length: 10

EvilBody1x
```

The generated string for the PCRE is only one out of a possibly infinite set of matching strings. This is the reason that the traffic generated by GENESIDS for a single rule might differ in different runs.

7.3.3 Limitations

GENESIDS is designed to only issue legal HTTP requests. Thus, no rules regarding another protocol are accepted.

While GENESIDS (same as Snort) also accepts hex encoded characters, it only allows the first 128 readable hex chars and `\n` and `\r`. If a rule contains a PCRE expression that is not accepted by the exrex command, GENESIDS will issue a warning and not produce any traffic for this rule.

The Snort database contains descriptions of almost all current attacks. Nevertheless, rare events or zero day exploits are not included and such an attack description for GENESIDS has to be manually created.

7.3.4 Generating Mixed Traffic

GENESIDS is designed to generate pure malicious traffic only. But there are numerous test scenarios where a mixed traffic set, consisting of attack traffic mixed with realistic benign traffic, is needed.

We propose to use a stateful L4-L7 traffic generator for the realistic benign part and mix this traffic with the attack traffic generated by GENESIDS. The stateful L4-L7 traffic generator that we use for our experiments is Cisco's TRex. TRex has been developed using Intel's DPDK library which allows kernel bypass methods for packet capturing. This enables timely precise and fast (up to 200 Gbit/s) network traffic generation. TRex is available as open-source software under the Apache license.[47] TRex statefully generates traffic including payload [151], according to the configuration in so called traffic templates. The main ingredients of a template are one or multiple pcap traces containing one TCP flow each. The template also contains instructions about how many flow Connections Per Second (CPS) should be generated for each pcap trace. TRex already ships with different examples of realistic traffic, but such templates can also be created manually with own pcap traces. The traffic traces shipped with TRex contain benign traffic only.

It is important to emphasize that there is a critical difference, especially for network monitoring appliances like NIDS, if the same traffic is replayed using different Packets Per Second (PPS) rates or CPS rates. Most traffic replaying utilities (e.g., tcpreplay) use increasing PPS rates. To replay traffic at higher rates such utilities simple reduce the inter-packet time and, thus, the traffic gets basically compressed in time. The problem is, that this is not what is encountered in reality in high-speed networks. Here, we have the same inter-packet delay between packets than in slower networks, the difference is that there are a myriad of concurrent connections to analyze. This is more realistically represented by increasing the CPS rate. Here, one

[47] `https://github.com/cisco-system-traffic-generator/trex-core`

connection gets replayed multiple times per second without changing the connection in the time domain. This is a different kind of challenge for NIDS, because now they have to cope with a high number of simultaneous packets, belonging to different connections at the same time. This is much more demanding, as data belonging to different caches has to be analyzed and, thus, the system is put under more stress.

The stateful generation of traffic is conducted in the following way: TRex initially reads the configuration file and loads all contained pcap traces to memory. Then it starts with the first request from every pcap trace and sends it out on the configured egress Network Interface Controller (NIC). The routing device or switch receiving this packet is routing the packets to the ingress NIC of the TRex device. As TRex receives the request packet, it responds by sending the corresponding response packet from the pcap trace on the egress NIC. The switch again routes this packet to the TRex ingress NIC and the same operation is repeated with the next request and response packets of the trace. This process is started multiple times per second per pcap trace depending on the configured CPS of the pcap trace. To include a NIDS in this scenario one port of the switch is configured as mirroring device, forwarding a copy of every packet to the NIDS under test. The source and destination addresses of the packets generated by TRex are defined in the traffic template and belong to a defined source and destination network, making routing possible in the first place.

Our proposed scenario for generating mixed traffic with GENESIDS and TRex is sketched in Figure 7.1. In a first step, GENESIDS is used to generate attack traffic according to the given Snort rules. This traffic is captured and split in single pcap traces containing one TCP flow per attack. This is facilitated by the fact that GENESIDS uses a completely new TCP connection for every attack. In a second step these pcap traces are included in the template together with benign traffic. The benign traffic can either consist of own handcrafted traces or it can consist of one of the benign traffic traces shipped with TRex. The proportion of benign and malicious traffic depends heavily on the application scenario. Finally, this traffic template is used with TRex to generate mixed traffic.

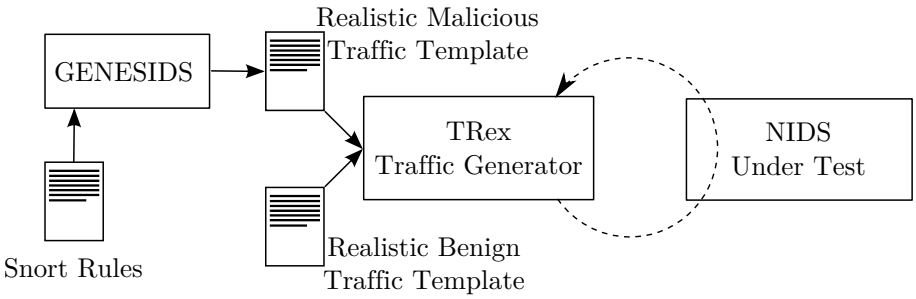


Figure 7.1 – Architecture of how to create a mixed traffic set with GENESIDS and TRex; derived from [29].

7.4 Evaluation

In this section we present experiments that we conducted to assess if GENESIDS reliably generates the attack traffic according to the given attack description and if the generated attack traffic triggers the expected events in a NIDS. All the configuration files and attack traces used in this experiments are available online.[48]

To have realistic and up-to-date attack definitions we used the following Snort signature databases (as of January 22th 2018):

- Rules provided to Snort.org subscribers (Snapshot 29111);
- community rule-set from Snort.org; and
- all rules as provided by Emerging Threats.[49]

From the above rules we used all HTTP-related rules accepted by GENESIDS. This results in a rule-set consisting of 8101 different rules.

The following experiment consisted of two steps: In a first step we generated attack traffic with GENESIDS for the above 8101 rules. Then we captured this traffic with *tcpdump*[50] and stored it for later usage. In the second step this traffic was analyzed with Snort (version 2.9.11, built from source). The baseline assumption is that Snort should trigger the corresponding event for all of the generated HTTP requests.

The first step, generating the attack traffic with GENESIDS and capturing it, is depicted in Figure 7.2. As described in Section 7.3, GENESIDS generates one HTTP request for every Snort signature from the rule-set. The request is sent to an HTTP server (in our case an Apache server version 2.4.10). The most common response of this server is a *404, Not Found* because almost all resources requested are not available in the default configuration as used with our Apache server.

The second step, analyzing the traffic with Snort, was conducted the following way: We configured Snort to run in IDS mode, and analyzed the captured pcap file, containing the attack traffic generated by GENESIDS, using the exact same rule-set that GENESIDS used for traffic generation. We used the default Snort configuration file

[48] `http://www.ccs-labs.org/~erlacher/resources/`
[49] `rules.emergingthreats.net/open/snort-2.9.0/emerging-all.rules`
[50] `tcpdump.org`

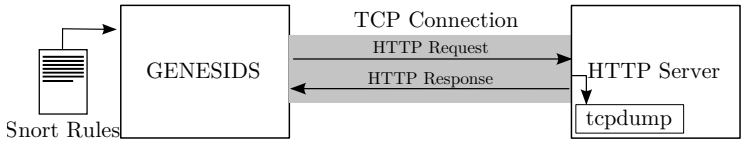


Figure 7.2 – Setup for the traffic generation experiments; derived from [29].

as shipped with the installation archive, the only relevant changes were increasing the logged alerts per packet to be able to see all triggered events. We also used the `-k none` switch to make Snort also accept packets with checksum errors.

During analysis, all events triggered by Snort were written to the event file. The event file contains one entry per triggered event. An entry contains, among other information, the SID of the rule that triggered this event, the source and destination address, and the source and destination port of the corresponding TCP connection. Because GENESIDS uses one TCP connection per generated request, a tuple consisting of the event TCP source port and SID is unique if during generation of the trace no TCP port has been used twice. If this applies, events from the Snort event file can easily be assigned to the corresponding HTTP request generated by GENESIDS.

The experiment results are shown in Figure 7.3. We repeated this experiment 100 times to make sure that the results are representable. The reason for this is, that, as explained before, many rules contain a RegEx pattern which will generate different attack traffic in different experiment runs.

GENESIDS generated exactly 8101 HTTP requests (one for every rule) in all experiment runs. We can confirm that GENESIDS used different TCP source ports for all TCP connections over all runs. Comparing the generated TCP port-SID tuples of the sent HTTP requests with the port-SID tuples of the events triggered by Snort, shows that more than 97 % of the generated attack requests triggered the corresponding event. As can be seen, on average only 223 (2.8 %) out of 8101 generated events triggered a false negative, e.g., the generated event did not trigger the correct alert. This confirms that the attack HTTP requests generated by GENESIDS trigger the correct event in almost all cases.

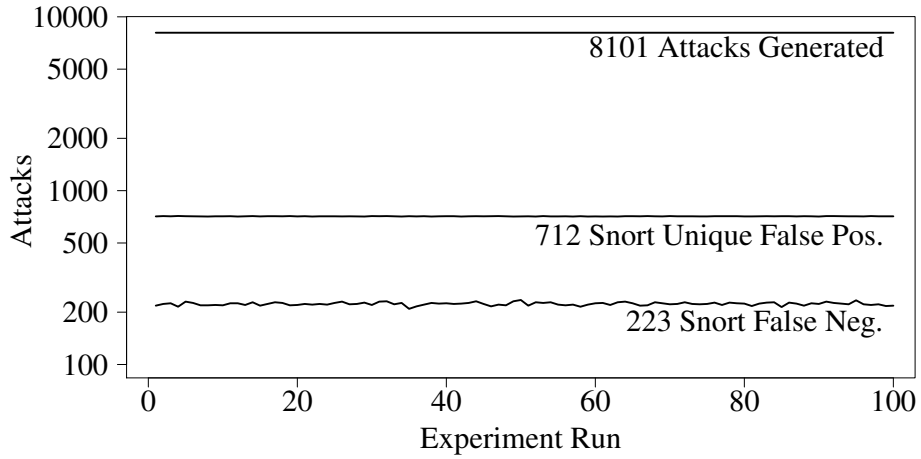


Figure 7.3 – Attacks generated by GENESIDS compared to the triggered events of Snort analyzing the attack traffic. Results of 100 experiment repetitions. Please note the logarithmic y-axis; derived from [29].

Snort also triggered 2847 false-positive alerts on average. False positives, in this case, denote alerts that where triggered by a generated attack request, that was not supposed to trigger this exact event. Please note, that one packet or request can possibly trigger multiple events. It has to be emphasized that the majority of these events has been triggered by only a handful of rules. On average the 2847 false-positive events have been triggered by only 712 different rules, and more than 41 % of these events were triggered by only two rules. This behavior is normal and is caused by so called "rule overlapping", as stated in a study by Massicotte and Labiche [152]. For evaluation reasons we applied as many rules as possible. In a production environment, the applied Snort rules are chosen carefully to firstly avoid false-positive alerts and secondly, because a high number of rules has a negative impact on the packet throughput performance.

Finally, we investigate the HTTP requests that did not trigger the corresponding event at least once in 100 runs. On average there were 223 HTTP requests causing these false negatives. Over 100 runs there were 363 unique rules causing a false negative at least once. We manually inspected all of these rules and divided them in two categories: On one side we have the rules that always caused a false negative over all 100 runs, and on the other side we have the rules that triggered a true positive at least once.

Not surprisingly, rules from the latter category all contain PCRE encoded content patterns which cause a non-deterministic traffic generation, while rules from the former category must have a basic issue preventing correct traffic generation.

The category with rules never triggering the correct alert contains 179 rules. The reasons for not generating the correct traffic were the following:

- 61 contain a PCRE encoded content pattern which is enclosed by ^ and $. This entails that this string is expected to be in a line by its own. But during traffic generation GENESIDS concatenates all patterns (PCRE encoded or not) in one line, thus, during analysis by Snort such patterns do not match anymore.

- 28 rules contain patterns with multiple line breaks (\r\n). Because GENESIDS / libcurl only generate standard compliant HTTP requests, multiple line breaks are ignored in cases where the standard does not expect them. The same holds if a rule pattern includes a line break at the beginning of the HTTP URI.

- 18 rules define multiple HTTP headers in one content field. This will not work because, as required by the standard, GENESIDS requires a line break in between single headers.

- 15 rules contain a PCRE pattern with the unsupported word character \w. The exrex command used by GENESIDS ignores such patterns and, thus, the generated request does not match the rule.

- 14 rules contain patterns with a \\ in a URI. GENESIDS generates them correctly, but Snort normalizes this to a single \ and, thus, the rule does not match anymore.
- 6 rules contain a pattern defined to be at the start of the client body. Because Snort applies rules to the client body only if the size of the body is greater than 5 characters, GENESIDS fills the client body with 5 characters before adding other client body data from rule patterns and, thus, these rules do not match.
- 6 rules contain patterns with the word boundary anchor \b in a PCRE at the beginning of a URI pattern where another URI pattern is added first, without any non-word character in between. This results in the rule not matching the produced HTTP request.
- 5 rules contain patterns with the character # in a URI. This character is reserved according to Berners-Lee et al. [169] and, thus, ignored by GENESIDS.
- 5 rules contain patterns with the disallowed [169] character + in a URI. GENESIDS adds this character literally but Snort interprets it as a whitespace character, causing a false negative.
- The remaining 21 rules have similar singular problems, which we do not describe individually.

The second category of rules, which triggered at least one false negative and at least one true positive alert in 100 runs, contains 184 rules. As stated above, all rules contain at least one PCRE pattern. The reason that these rules sometimes fail and sometimes not is the non-deterministic generation of traffic from PCRE patterns.

- 123 of these rules contain a PCRE pattern including a character class starting with a negation. For example [^\x2F] which translates to "every character but a /". Sometimes such patterns produce an unsupported character which leads to the generated traffic not matching the rule anymore.
- 46 rules contain a PCRE pattern whit a single . character. Again, in some cases such patterns produce an unsupported character.

  Here we have to emphasize that GENESIDS already replaces the most common occurrences of [^ patterns and . character combinations and that the cases described above are the very few instances which have not been replaced.
- The remaining 15 rules contain a PCRE pattern which includes multiple alternation (separated with the | character). At least one of the alternations contains one of the problematic patterns described int the first category of false negatives, which leads to the generated traffic not matching the rule.

With the result of these experiments we conclude that GENESIDS accurately generates malicious traffic according to the given attack definitions. The generated traffic reliably triggers the corresponding event in the NIDS Snort.

7.5 Lessons Learned

In this chapter we presented our novel attack traffic generator GENESIDS. Using the Snort signature syntax format for attack description, it is not only able to accept self written attacks but takes advantage of thousands up-to-date and community maintained attack definitions. In our experiments we showed that more than 97 % of the 8101 different generated attacks triggered the corresponding event in the NIDS Snort, confirming that GENESIDS reliably generates the network traffic according to the given attack descriptions. While we showed how GENESIDS can be used for testing signature-based NIDS, we are confident that it can also be used for the evaluation of anomaly-based NIDS. This chapter successfully answers research question four: How can we automatically generate malicious test traffic for NIDS, which includes a representable and comprehensive set of attacks?

Chapter 8

Conclusion

WITH this PhD thesis we contributed to solve the problem of efficient intrusion detection in high-speed networks. We proposed several methods to increase the efficiency in terms of detection accuracy and traffic throughput capability.

Firstly, we motivated this work and presented basic technologies and concepts, as well as related work in this domain.

In Chapter 3, we familiarized the reader with the threats that have been introduced by the technologies that enable the Web 2.0. Hereby, we also analyzed current mitigation techniques and addressed open research problems in this field. We confirmed that the relatively low network throughput of Network Intrusion Detection Systems (NIDS), compared to today's high network traffic rates, is one of the most pressing ones.

In Chapter 4 we increased the detection accuracy of anomaly-based NIDS by combining multiple Anomaly Detection Algorithms (ADAs) on a single machine. The combination of multiple algorithms generates high computational load, which usually leads to packet drops and, thus, intrusions that can not be detected. To mitigate this, we added a load allocation scheme which observes the load of the single ADAs and makes packets skip overloaded instances of these algorithms. Therefore, packets can still be analyzed by all other algorithms and, thus, intrusions can still be detected. In the evaluation we showed that the detection accuracy benefits from a combination of multiple ADAs compared to the detection accuracy of single instances of the combined ADAs. The resulting high load can be mitigated with our novel load allocation scheme, leading to a higher network throughput rate. This answered our first research question: How can we combine multiple ADAs on a single machine, mitigating the negative impact of the high computational load, caused by multiple ADAs?

In Chapter 5 we increase the network throughput performance for network monitoring in general and for Flow-based intrusion detection in particular, by proposing

two methods for preprocessing the Hypertext Transfer Protocol (HTTP) before analysis. Firstly, we showed how to include important parts of HTTP as Information Element (IE) fields into the Internet Protocol Flow Information Export (IPFIX) protocol. This significantly reduces the data portion of HTTP to be analyzed by network monitoring appliances. We showed that our purposely build HTTP parser can keep up with state-of-the-art HTTP parsing appliances and even outperforms them in some functionality metrics. We proposed the HTTP-related IE fields to the standardizing body for IPFIX, the Internet Assigned Numbers Authority (IANA). By now, our proposed fields have been accepted and are part of the IPFIX standard. Secondly, we proposed a filtering approach called HTTP-based Payload Aggregation (HPA). It significantly reduces the amount of HTTP data and exports this data in form of packets. Thus, traditional, packet-based network monitoring appliances and NIDS can use this data without further adaptions. In our evaluation experiments we showed that our approach reduces the analyzed HTTP data to 2.5 % of its original size. When analyzing this data with the NIDS Snort, still 97 % of the originally contained intrusions can be detected, showing a speedup in the network throughput rate of 44 %. This significantly increases the throughput performance of existing packet-based NIDS. For both of the described approaches we chose HTTP as an example for a modern, interleaving application-layer protocol. We strongly believe that the presented approaches can be applied to other application-layer protocols as well. This chapter answered research question two: How to reduce the amount and aggregate the interesting parts of HTTP traffic for network monitoring?

In Chapter 6 we presented our novel, signature-based Flow-based NIDS FIXIDS. It takes advantage of standardized, HTTP-related IPFIX IE fields and uses them for intrusion detection. For attack descriptions FIXIDS uses the signature syntax of the popular NIDS Snort and accepts all its HTTP-related signatures. This ensures that thousands of up-to-date and community validated signatures are available. By performing intrusion detection on IPFIX Flows using standardized IEs, we guarantee that FIXIDS can directly accept Flows from switches with Flow exporting capabilities supporting these standardized fields. Our evaluation experiment results show that FIXIDS has the same, very high detection accuracy as Snort, but at a much higher network throughput rate. When analyzing the same HTTP traffic, a NIDS environment using FIXIDS can cope with four times the traffic throughput rate compared to Snort. We showed that, when FIXIDS is deployed in an existing intrusion detection scenario, which analyzes HTTP and non-HTTP traffic, FIXIDS takes away a substantial part of the load and, thus, considerably increases the overall network throughput rate of the system. This answered the third research question: Can we use HTTP-enriched IPFIX Flows for efficient intrusion detection?

In Chapter 7 we proposed an automatic traffic generator for malicious HTTP traffic called GENESIDS. GENESIDS accepts attack descriptions in the form of Snort signatures

and, thus, thousands of up-to-date and community validated attack description are available. This guarantees an extraordinary variety of realistic malicious traffic and represents an unprecedented possibility to thoroughly test novel NIDS. In our evaluation experiments we showed that GENESIDS is able to reliably generate attack traffic for more than 8000 different Snort signatures, thereby triggering the correct alert in more than 97 % of the cases. For now, GENESIDS only generates HTTP related attacks. This was enough for the purposes of this thesis, but it would be beneficial for the NIDS community to extend the traffic generation capability to more application-layer protocols. This answered the fourth and last research question: How can we automatically generate malicious test traffic for NIDS which includes a representable and comprehensive set of attacks?

In summary, this PhD thesis gives multiple contributions to increase the detection accuracy and network throughput of NIDS.

The most important and lasting contribution is the NIDS FIXIDS, which was developed by taking advantage of several of our previous works. As pointed out earlier, more and more network monitoring appliance manufacturers are including application-layer specific information into the exported Flows. This paves the way for a more widespread application of Flow-based signature-based intrusion detection, not only on HTTP as with FIXIDS, but on all available application layer related Flow fields.

From today's perspective, promising improvements in intrusion detection can be expected from the distribution of NIDS at different points of a network and the combination of the results thereof. Different vantage points allow a more specific and efficient adaption of detection methods to the expected traffic. The results of the single NIDS could, e.g., be used to whitelist traffic in real-time at other locations in the network, thus, freeing resources which enables higher throughput rates.

List of Abbreviations

ADA	Anomaly Detection Algorithm
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASLR	Address Space Layout Randomization
AV software	Anti-Virus software
BMBF	Federal Ministry for Education and Research (Bundesministerium für Bildung und Forschung)
BOM	Byte Order Mark
CA	Certification Authority
CCC	Command and Control Communication
CDF	Cumulative Distribution Function
CPS	Connections Per Second
CPU	Central Processing Unit
CSP	Content Security Policy
CSRF	Cross Site Request Forgery
CVE	Common Vulnerability and Exposure
CVSS	Common Vulnerability Scoring System
DARPA	Defense Advanced Research Projects Agency
DASH	Dynamic Adaptive Streaming over HTTP
DDOS	Distributed Denial of Service
DFA	Deterministic Finite Automaton
DHTML	Dynamic HTML
DNS	Domain Name System
DOS	Denial of Service
DPA	Dialog-based Payload Aggregation
DPI	Deep Packet Inspection
DSL	Domain Specific Language
FBI	Federal Bureau of Investigation
FIXIDS	IPFIX-based Signature-based Intrusion Detection System
FOSS	Free and Open Source Software

FPA	Front Payload Aggregation
GPL	GNU General Public License
HIDS	Host-based Intrusion Detection System
HLS	HTTP Live Streaming
HPA	HTTP-based Payload Aggregation
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IC3	Internet Crime Complaint Center
IDS	Intrusion Detection System
IE	Information Element
IETF	Internet Engineering Task Force
IPFIX	Internet Protocol Flow Information Export
IPS	Intrusion Prevention System
IRC	Internet Relay Chat
ISP	Internet Service Provider
ITU	International Telecommunication Union
MTU	Maximum Transmission Unit
NETAD	Network Traffic Anomaly Detector
NGFW	Next Generation Firewall
NIC	Network Interface Controller
NIDS	Network Intrusion Detection System
OS	Operating System
PCRE	Perl Compatible Regular Expression
PDU	Protocol Data Unit
PHAD	Packet Header Anomaly Detection
PKI	Public Key Infrastructure
PPS	Packets Per Second
RegEx	Regular Expression
RFC	Request for Comments
SIMD	Single Instruction Multiple Data
SMTP	Simple Mail Transfer Protocol
SOP	Same Origin Policy
SSH	Secure Shell
SVG	Scalable Vector Graphics
SWG	Secure Web Gateway
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDHR	Universal Declaration of Human Rights
UN	United Nations

URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTM	Unified Threat Management
Vermont	VERsatile MONitoring Toolkit
W3C	World Wide Web Consortium
WAF	Web Application Firewall
WG	Working Group
www	World Wide Web
XML	Extensible Markup Language
XSS	Cross Site Scripting

List of Figures

List of Tables

Bibliography

[1] ITU, Measuring the Information Society Report 2017. International Telecommunication Union, Nov. 2017, vol. 1.

[2] M. Trevisan, D. Giordano, I. Drago, M. Mellia, and M. Munafo, "Five Years at the Edge: Watching Internet from the ISP Network," in *14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2018))*, Heraklion, Greece: ACM, Dec. 2018.

[3] Symantec, "Internet Security Threat Report," Symantec Corporation, Tech. Report vol. 23, Mar. 2018.

[4] S. T. Zargar, J. Joshi, and D. Tipper, "A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, 2046–2069, Nov. 2013.

[5] J. J. Santanna, R. van Rijswijk-Deij, R. Hofstede, A. Sperotto, M. Wierbosch, L. Z. Granville, and A. Pras, "Booters - An Analysis of DDoS-As-A-Service Attacks," in *International Symposium on Integrated Network Management (IFIP/IEEE IM 2015)*, Ottawa, Canada: IEEE, May 2015, 243–251.

[6] R. Brewer, "Ransomware Attacks: Detection, Prevention and Cure," *Network Security*, vol. 2016, no. 9, 5–9, Sep. 2016.

[7] R. Kemmerer and G. Vigna, "Intrusion Detection: A Brief History and Overview," *IEEE Computer, Special Issue on Security and Privacy*, pp. 27–30, Apr. 2002.

[8] R. Bray, D. Cid, and A. Hay, OSSEC Host-based Intrusion Detection Guide. Syngress, 2008.

[9] H. Debar, M. Dacier, and A. Wespi, "Towards a Taxonomy of Intrusion-Detection Systems," *Elsevier Computer Networks*, vol. 31, no. 8, pp. 805–822, Apr. 1999.

[10] S. Axelsson, "Intrusion Detection Systems: A Survey and Taxonomy," Chalmers University Gothenburg, Tech. Rep. 99-15, 2000.

[11] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Macia-Fernandez, and E. Vazquez, "Anomaly-based Network Intrusion Detection: Techniques, Systems and Challenges," *Computers & Security*, vol. 28, no. 1-2, 18–28, Feb. 2009.

[12] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network Anomaly Detection: Methods, Systems and Tools," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, 303–336, Jun. 2014.

[13] R. Sommer and V. Paxson, "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection," in *IEEE Symposium on Security and Privacy (SP 2010)*, Berkeley, CA: IEEE, May 2010, 305–316.

[14] S. Biles, "Detecting the Unknown with Snort and the Statistical Packet Anomaly Detection Engine (SPADE)," Computer Security Online Ltd., Tech. Rep., 2006.

[15] K. Wang and S. J. Stolfo, "Anomalous Payload-based network intrusion detection," in *7th International Symposium on Recent Advances in Intrusion Detection (RAID 2004)*, Sophia Antipolis, France: Springer, Sep. 2004, pp. 203–222.

[16] C.-J. Chung, P. Khatkar, T. Xing, J. Lee, and D. Huang, "NICE: Network Intrusion Detection and Countermeasure Selection in Virtual Network Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 4, 198–211, Jul. 2013.

[17] P.-C. Lin and J.-H. Lee, "Re-examining the Performance Bottleneck in a NIDS with Detailed Profiling," *Journal of Network and Computer Applications*, vol. 36, no. 2, 768–780, Mar. 2013.

[18] M. Roesch, "Snort: Lightweight Intrusion Detection for Networks," in *13th USENIX Conference on System Administration (LISA 1999)*, Seattle, WA, Nov. 1999, pp. 229–238.

[19] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Elsevier Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, Dec. 1999.

[20] K. Hwang, M. Cai, Y. Chen, and M. Qin, "Hybrid Intrusion Detection with Weighted Signature Generation over Anomalous Internet Episodes," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 1, pp. 41–55, Jan. 2007.

[21] G. Kim, S. Lee, and S. Kim, "A Novel Hybrid Intrusion Detection Method Integrating Anomaly Detection with Misuse Detection," *Expert Systems with Applications*, vol. 41, no. 4, 1690–1700, Mar. 2014.

[22] B. Stritter, F. Freiling, H. König, R. Rietz, S. Ullrich, A. von Gernler, F. Erlacher, and F. Dressler, "Cleaning up Web 2.0's Security Mess - at Least Partly," *IEEE Security & Privacy*, vol. 14, no. 2, pp. 48–57, Mar. 2016.

[23] M. Berger, F. Erlacher, C. Sommer, and F. Dressler, "Adaptive Load Allocation for Combining Anomaly Detectors Using Controlled Skips," in *3rd IEEE International Conference on Computing, Networking and Communications (ICNC 2014), CNC Workshop*, Honolulu, HI: IEEE, Feb. 2014, pp. 792–796.

[24] F. Erlacher, W. Estgfaeller, and F. Dressler, "Improving Network Monitoring Through Aggregation of HTTP/1.1 Dialogs in IPFIX," in *41st IEEE Conference on Local Computer Networks (LCN 2016)*, Dubai, UAE: IEEE, Nov. 2016, pp. 543–546.

[25] F. Erlacher and F. Dressler, "High Performance Intrusion Detection Using HTTP-based Payload Aggregation," in *42nd IEEE Conference on Local Computer Networks (LCN 2017)*, Singapore: IEEE, Oct. 2017, pp. 418–425.

[26] F. Erlacher and F. Dressler, "FIXIDS: A High-Speed Signature-based Flow Intrusion Detection System," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2018)*, Taipei, Taiwan: IEEE, Apr. 2018.

[27] F. Erlacher and F. Dressler, "On High-Speed Flow-based Intrusion Detection using Snort-compatible Signatures," *IEEE Transactions on Dependable and Secure Computing*, submitted.

[28] F. Erlacher and F. Dressler, "How to Test an IDS? GENESIDS: An Automated System for Generating Attack Traffic," in *ACM SIGCOMM 2018, Workshop on Traffic Measurements for Cybersecurity (WTMC 2018)*, Budapest, Hungary: ACM, Aug. 2018, pp. 46–51.

[29] F. Erlacher and F. Dressler, "Testing IDS using GENESIDS: Realistic Mixed Traffic Generation for IDS Evaluation," in *ACM SIGCOMM 2018, Demo Session*, Budapest, Hungary: ACM, Aug. 2018, pp. 153–155.

[30] F. Erlacher, "Network Monitoring for Todays Internet," in *International Conference on Networked Systems (NetSys 2015), PhD Forum*, Cottbus, Germany, Mar. 2015.

[31] F. Erlacher, S. Woertz, and F. Dressler, "A TLS Interception Proxy with Real-Time Libpcap Export," in *41st IEEE Conference on Local Computer Networks (LCN 2016), Demo Session*, Dubai, UAE: IEEE, Nov. 2016.

[32] M. Segata, B. Bloessl, S. Joerer, F. Erlacher, M. Mutschlechner, F. Klingler, C. Sommer, R. Lo Cigno, and F. Dressler, "Shadowing or Multi-Path Fading: Which Dominates in Inter-Vehicle Communication?" University of Innsbruck, Institute of Computer Science, Technical Report CCS-2013-03, Jun. 2013.

[33] F. Erlacher, F. Klingler, C. Sommer, and F. Dressler, "On the Impact of Street Width on 5.9 GHz Radio Signal Propagation in Vehicular Networks," in *11th IEEE/IFIP Conference on Wireless On demand Network Systems and Services (WONS 2014)*, Obergurgl, Austria: IEEE, Apr. 2014, pp. 143–146.

[34] M. Mutschlechner, F. Klingler, F. Erlacher, F. Hagenauer, M. Kiessling, and F. Dressler, "Reliable Communication using Erasure Codes for Monitoring Bats in the Wild," in *33rd IEEE Conference on Computer Communications (INFOCOM 2014), Student Activities*, Toronto, Canada: IEEE, Apr. 2014, pp. 189–190.

[35] F. Erlacher, B. Weber, J.-T. Fischer, and F. Dressler, "AvaRange - Using Sensor Network Ranging Techniques to Explore the Dynamics of Avalanches," in *12th IEEE/IFIP Conference on Wireless On demand Network Systems and Services (WONS 2016)*, Cortina d'Ampezzo, Italy: IEEE, Jan. 2016, pp. 120–123.

[36] F. Erlacher, F. Dressler, and J.-T. Fischer, "New Insights on a Sensor Network based Measurement Platform for Avalanche Dynamics," in *International Snow Science Workshop (ISSW 2018)*, Innsbruck, Austria, Oct. 2018, pp. 31–34.

[37] S. Lee, K. Levanti, and H. S. Kim, "Network Monitoring: Present and Future," *Elsevier Computer Networks*, vol. 65, 84–98, Jun. 2014.

[38] J. Morsink, The Universal Declaration of Human Rights: Origins, Drafting, and Intent, English. University of Pennsylvania Press, 1999.

[39] G. Greenwald, No Place to Hide: Edward Snowden, the NSA, and the US Surveillance State, English. Picador, 2014.

[40] A. Hintz and L. Dencik, "The Politics of Surveillance Policy: UK Regulatory Dynamics After Snowden," *Internet Policy Review*, vol. 5, no. 3, Sep. 2016.

[41] G. Orwell, Nineteen Eighty-Four, English, ser. Twentieth century classics. London: Secker and Warburg, 1949.

[42] J. Zittrain and B. Edelman, "Internet Filtering in China," *IEEE Internet Computing*, vol. 7, no. 2, 70–77, Mar. 2003.

[43] Y. Chen and A. S. Cheung, "The Transparent Self Under Big Data Profiling: Privacy and Chinese Legislation on the Social Credit System," *The Journal of Comparative Law*, vol. 12, no. 2, pp. 356–378, Jun. 2017.

[44] A. R. Jonsen, The Birth of Bioethics. Oxford University Press, 2003.

[45] G. Bianchi, E. Boschi, D. I. Kaklamani, E. Koutsoloukas, G. V. Lioudakis, F. Oppedisano, M. Petraschek, F. Ricciato, and C. Schmoll, "Towards Privacy-Preserving Network Monitoring: Issues and Challenges," in *18th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 2007)*, Athens, Greece: IEEE, Sep. 2007.

[46] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "SEPIA: Privacy-preserving Aggregation of Multi-domain Network Events and Statistics," in *19th USENIX Conference on Security (Security'10)*, Washington, DC: USENIX, Aug. 2010.

[47] E. Toch, C. Bettini, E. Shmueli, L. Radaelli, A. Lanzi, D. Riboni, and B. Lepri, "The Privacy Implications of Cyber Security Systems: A Technological Survey," *ACM Computing Surveys*, vol. 51, no. 2, Jun. 2018.

[48] S. Degli Esposti, V. Pavone, and E. Santiago-Gomez, "Aligning Security and Privacy: The Case of Deep Packet Inspection," in *Surveillance, Privacy and Security: Citizens' Perspectives*, M. Friedwald, P. Burgess, J. Cas, R. Bellanova, and W. Peissl, Eds., Routledge, 2017.

[49] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han, "DFC: Accelerating String Pattern Matching for Network Applications," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2016)*, Santa Clara, CA: USENIX, Mar. 2016, 551–565.

[50] C. Stylianopoulos, M. Almgren, O. Landsiedel, and M. Papatriantafilou, "Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization," in *46th International Conference on Parallel Processing (ICPP 2017)*, Bristol, UK: IEEE, Aug. 2017, 472–482.

[51] A. Backurs and P. Indyk, "Which Regular Expression Patterns are Hard to Match?" In *57th Annual Symposium on Foundations of Computer Science (FOCS 2016)*, New Brunswick, NJ: IEEE, Oct. 2016, 457–466.

[52] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, vol. 5230, Cambridge, MA: Springer, Sep. 2008, pp. 116–134.

[53] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast Regular Expression Matching Using Small TCAMs for Network Intrusion Detection and Prevention Systems," in *19th USENIX Conference on Security (USENIX Security 2010)*, Washington, DC: ACM, Aug. 2010, pp. 8–23.

[54] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975.

[55] C. Xu, S. Chen, J. Su, S.-M. Yiu, and L. C. Hui, "A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, 2991–3029, May 2016.

[56] R. Antonello, S. Fernandes, D. Sadok, J. Kelner, and G. Szabo, "Design and Optimizations for Efficient Regular Expression Matching in DPI Systems," *Computer Communications*, vol. 61, 103–120, May 2015.

[57] M. Becchi and S. Cadambi, "Memory-Efficient Regular Expression Search using State Merging," in *26th International Conference on Computer Communications (INFOCOM 2007)*, Barcelona, Spain: IEEE, May 2007, 1064–1072.

[58] J. M. Silva, P. Carvalho, and S. R. Lima, "Inside Packet Sampling Techniques: Exploring Modularity to Enhance Network Measurements," *International Journal of Communication Systems*, vol. 30, no. 6, Mar. 2017.

[59] T. Zseby, M. Molina, N. Duffield, S. Niccolini, and F. Raspall, "Sampling and Filtering Techniques for IP Packet Selection," IETF, RFC 5475, Mar. 2009.

[60] T. Dietz, B. Claise, P. Aitken, F. Dressler, and G. Carle, "Information Model for Packet Sampling Exports," IETF, RFC 5477, Mar. 2009, draft-ietf-psamp-info.

[61] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina, "Impact of Packet Sampling on Anomaly Detection Metrics," in *6th ACM SIGCOMM Internet Measurement Conference (IMC 2006)*, Rio de Janeiro, Brazil: ACM, Oct. 2006, 159–164.

[62] A. Pescape, D. Rossi, D. Tammaro, and S. Valenti, "On the Impact of Sampling on Traffic Monitoring and Analysis," in *22nd International Teletraffic Congress (ITC 2010)*, Amsterdam, Netherlands: IEEE, Sep. 2010.

[63] L. Braun, G. Muenz, and G. Carle, "Packet Sampling for Worm and Botnet Detection in TCP Connections," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2010)*, Osaka, Japan: IEEE, Apr. 2010, 264–271.

[64] N. Tsikoudis, A. Papadogiannakis, and E. P. Markatos, "LEoNIDS: A Low-Latency and Energy-Efficient Network-Level Intrusion Detection System," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 1, pp. 142–155, Jan. 2016.

[65] S. Kornexl, V. Paxson, H. Dreger, R. Sommer, and A. Feldmann, "Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic," in *5th ACM SIGCOMM Conference on Internet Measurement (IMC 2005)*, Berkeley, CA: ACM, Oct. 2005, pp. 267–272.

[66] T. Limmer and F. Dressler, "Flow-based Front Payload Aggregation," in *34th IEEE Conference on Local Computer Networks (LCN 2009): 4th IEEE LCN Workshop on Network Measurements (WNM 2009)*, Zurich, Switzerland: IEEE, Oct. 2009, pp. 1102–1109.

[67] T. Limmer and F. Dressler, "Improving the Performance of Intrusion Detection using Dialog-based Payload Aggregation," in *30th IEEE Conference on Computer Communications (INFOCOM 2011), 14th IEEE Global Internet Symposium (GI 2011)*, Shanghai, China: IEEE, Apr. 2011, pp. 833–838.

[68] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing," IETF, Tech. Rep. 7230, Jun. 2014.

[69] M. Belshe, M. Thomson, and R. Peon, "Hypertext Transfer Protocol Version 2 (HTTP/2)," IETF, RFC 7540, May 2015.

[70] W. Cherif, Y. Fablet, E. Nassor, J. Taquet, and Y. Fujimori, "DASH Fast Start Using HTTP/2," in *25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2015)*, Portland, OR, Mar. 2015, 25–30.

[71] R. Pantos and W. May, "HTTP Live Streaming," IETF, Tech. Rep. 88216, Aug. 2017.

[72] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow Monitoring Explained: From Packet Capture to Data Analysis with Netflow and IPFIX," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037–2064, May 2014.

[73] N. Brownlee, C. Mills, and G. Ruth, "Traffic Flow Measurement: Architecture," IETF, RFC 2722, Oct. 1999.

[74] B. Claise, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information," IETF, RFC 5101, Jan. 2008.

[75] J. Koegel, "One-way Delay Measurement based on Flow Data: Quantification and Compensation of Errors by Exporter Profiling," in *25th International Conference on Information Networking (ICOIN 2011)*, Kuala Lumpur, Malaysia: IEEE, Jan. 2011, 25–30.

[76] R. Hofstede, I. Drago, A. Sperotto, R. Sadre, and A. Pras, "Measurement Artifacts in Netflow Data," in *14th International Conference on Passive and Active Network Measurement (PAM 2013)*, Hong Kong, China: Springer, Mar. 2013.

[77] R. Hofstede, A. Pras, A. Sperotto, and G. D. Rodosek, "Flow-Based Compromise Detection: Lessons Learned," *IEEE Security & Privacy*, vol. 16, no. 1, 82–89, Jan. 2018.

[78] B. Claise, "Cisco Systems NetFlow Services Export Version 9," IETF, RFC 3954, Oct. 2004.

[79] N. Brownlee, "Flow-based Measurement: IPFIX Development and Deployment," *IEICE Transactions on Communications*, vol. 94, no. 8, 2190–2198, Aug. 2011.

[80] B. Trammell and E. Boschi, "An Introduction to IP Flow Information Export (IPFIX)," *IEEE Communications Magazine*, vol. 49, no. 4, pp. 89–95, Apr. 2011.

[81] J. Quittek, S. Bryant, B. Claise, P. Aitken, and J. Meyer, "Information Model for IP Flow Information Export," IETF, RFC 5102, Jan. 2008.

[82] A. Sperotto and A. Pras, "Flow-based Intrusion Detection," in *12th International Symposium on Integrated Network Management (IM 2011)*, Dublin, Ireland: IEEE, May 2011, 958–963.

[83] M. F. Umer, M. Sher, and Y. Bi, "Flow-Based Intrusion Detection: Techniques and Challenges," *Computers & Security*, vol. 70, 238–254, Sep. 2017.

[84] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An Overview of IP Flow-based Intrusion Detection," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 3, 343–356, Apr. 2010.

[85] D. Douglas, J. J. Santanna, R. de Oliveira Schmidt, L. Z. Granville, and A. Pras, "Booters: Can Anything Justify Distributed Denial-of-Service (DDoS) Attacks for Hire?" *Journal of Information, Communication and Ethics in Society*, vol. 15, no. 01, 90–104, 2017.

[86] R. Sadre, A. Sperotto, and A. Pras, "The Effects of DDoS Attacks on Flow Monitoring Applications," in *18th IEEE/IFIP Network Operations & Management Symposium (NOMS 2012)*, Maui, HI: IEEE, Apr. 2012, 269–277.

[87] R. Hofstede, V. Bartos, A. Sperotto, and A. Pras, "Towards Real-time Intrusion Detection for NetFlow and IPFIX," in *9th International Conference on Network and Service Management (CNSM 2013)*, Zurich, Switzerland, Oct. 2013, 227–234.

[88] A. Wagner and B. Plattner, "Entropy Based Worm and Anomaly Detection in Fast IP Networks," in *14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE 2005)*, Linkoping, Sweden: IEEE, Jun. 2005, 172–177.

[89] G. Nychis, V. Sekar, D. G. Andersen, H. Kim, and H. Zhang, "An Empirical Evaluation of Entropy-based Traffic Anomaly Detection," in *8th ACM SIGCOMM Conference on Internet Measurement (IMC 2008)*, Vouliagmeni, Greece: ACM, Oct. 2008, 151–156.

[90] M. Ring, D. Landes, and A. Hotho, "Detection of Slow Port Scans in Flow-based Network Traffic," *PLOS ONE*, vol. 13, no. 9, Sep. 2018.

[91] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the Slammer Worm," *IEEE Security & Privacy*, vol. 99, no. 4, 33–39, Jul. 2003.

[92] F. Dressler, W. Jaegers, and R. German, "Flow-based Worm Detection using Correlated Honeypot Logs," in *15. GI/ITG Fachtagung Kommunikation in Verteilten Systemen (KiVS 2007)*, T. Braun, G. Carle, and B. Stiller, Eds., Bern, Switzerland: VDE, Feb. 2007, pp. 181–186.

[93] S. A. Abdulla, S. Ramadass, A. Altaher, and A. A. Nassiri, "Setting a Worm Attack Warning by Using Machine Learning to Classify Netflow Data," *International Journal of Computer Applications*, vol. 36, no. 2, 49–56, Dec. 2011.

[94] J. A. Dev, "Bitcoin Mining Acceleration and Performance Quantification," in *27th Canadian Conference on Electrical and Computer Engineering (CCECE 2014)*, Toronto, Canada: IEEE, May 2014.

[95] A. Karasaridis, B. Rexroad, and D. Hoeflin, "Wide-Scale Botnet Detection and Characterization," in *First Workshop on Hot Topics in Understanding Botnets (HotBots 2007)*, vol. 7, Cambridge, MA: Usenix, Apr. 2007.

[96] Z. Qiu, D. J. Miller, and G. Kesidis, "Flow Based Botnet Detection through Semi-Supervised Active Learning," in *42nd International Conference on Acoustics, Speech and Signal Processing (ICASSP 2017)*, New Orleans, LA: IEEE, Mar. 2017, 2387–2391.

[97] L. Hellemons, L. Hendriks, R. Hofstede, A. Sperotto, R. Sadre, and A. Pras, "SSHCure: A Flow-Based SSH Intrusion Detection System," in *6th International Conference on Autonomous Infrastructure, Management, and Security (AIMS 2012)*, Luxembourg, Luxembourg: Springer, Jun. 2012, pp. 86–97.

[98] R. Hofstede, L. Hendriks, A. Sperotto, and A. Pras, "SSH Compromise Detection using NetFlow/IPFIX," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, 20–26, Oct. 2014.

[99] R. Hofstede and L. Hendriks, "Unveiling SSHCure 3.0: Flow-based SSH Compromise Detection," in *International Conference on Networked Systems (NetSys 2015), Demo Session*, Cottbus, Germany, Mar. 2015.

[100] F. Dressler and G. Carle, "HISTORY - High Speed Network Monitoring and Analysis," in *24th IEEE Conference on Computer Communications (INFOCOM 2005), Poster Session*, Miami, FL: IEEE, Mar. 2005.

[101] 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 (MonAM 2006)*, Tübingen, Germany: IEEE, Sep. 2006, pp. 62–65.

[102] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," IETF, RFC 8446, Aug. 2018.

[103] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel, "Website Fingerprinting at Internet Scale," in *23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*, San Diego, CA: The Internet Society, Feb. 2016.

[104] P. Velan, M. Cermak, P. Celeda, and M. Dravsar, "A Survey of Methods for Encrypted Traffic Classification and Analysis," *International Journal of Network Management*, vol. 25, no. 5, 355–374, Jul. 2015.

[105] D. Flanagan, JavaScript: The Definitive Guide. O'Reilly Media, Inc., 2006.

[106] C. Castelluccia, "Behavioural Tracking on the Internet: A Technical Perspective," in *European Data Protection: In Good Health?* Springer, 2012, 21–33.

[107] V. Prevelakis and D. Spinellis, "Sandboxing Applications," in *USENIX Annual Technical Conference, FREENIX Track*, Boston, MA: USENIX, Jun. 2001, 119–126.

[108] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-Space Randomization," in *11th ACM Conference on Computer and Communications Security (CCS 2004)*, Washington, D.C.: ACM, Oct. 2004, 298–307.

[109] G. Lawton, "Web 2.0 Creates Security Challenges," *Computer*, vol. 40, no. 10, 13–16, Oct. 2007.

[110] J. Grossman, S. Fogie, R. Hansen, A. Rager, and P. D. Petkov, XSS Attacks: Cross Site Scripting Exploits and Defense. Syngress, 2007.

[111] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and Defenses," in *21st USENIX Security Symposium*, Bellevue, WA, Aug. 2012, 413–428.

[112] A. Sudhodanan, R. Carbone, L. Compagna, N. Dolgin, A. Armando, and U. Morelli, "Large-Scale Analysis & Detection of Authentication Cross-Site Request Forgeries," in *European Symposium on Security and Privacy (EuroS&P 2017)*, Paris, France: IEEE, Apr. 2017, 350–365.

[113] M. Handley, V. Paxson, and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-To-End Protocol Semantics," in *10th USENIX Security Symposium*, Washington, DC: Usenix, Aug. 2001, pp. 115–131.

[114] S. Rose, M. Larson, D. Massey, R. Austein, and R. Arends, "DNS Security Introduction and Requirements," IETF, RFC 4033, Mar. 2005.

[115] N. Leavitt, "Internet Security Under Attack: The Undermining of Digital Certificates," *Computer*, vol. 44, no. 12, 17–20, Dec. 2011.

[116] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens, "Automatic and Precise Client-Side Protection against CSRF Attacks," in *16th European Symposium on Research in Computer Security (ESORICS 2011)*, ser. Lecture Notes in Computer Science, vol. 6879, Leuven, Belgium: Springer, Sep. 2011, 100–116.

[117] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A Client-side Solution for Mitigating Cross-site Scripting Attacks," in *ACM Symposium on Applied Computing (SAC 2006)*, Dijon, France: ACM, Apr. 2006, pp. 330–337.

[118] P. Raman, "JaSPIn: JavaScript Based Anomaly Detection of Cross-site Scripting Attacks," PhD thesis, Carleton University, Ottawa, ON, Canada, 2008.

[119] T. Alexenko, M. Jenne, S. D. Roy, and W. Zeng, "Cross-Site Request Forgery: Attack and Defense," in *7th IEEE Consumer Communications and Networking Conference (CCNC 2010)*, Las Vegas, NV: IEEE, Jan. 2010.

[120] H. Shahriar and M. Zulkernine, "S2XS2: A Server Side Approach to Automatically Detect XSS Attacks," in *9th Conference on Dependable, Autonomic and Secure Computing (DASC 2011)*, Sydney, Australia: Springer, Dec. 2011, 7–14.

[121] M. Van Gundy and H. Chen, "Noncespaces: Using Randomization to Defeat Cross-Site Scripting Attacks," *Computers & Security*, vol. 31, no. 4, 612–628, Jun. 2012.

[122] T. Jim, N. Swamy, and M. Hicks, "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies," in *16th International Conference on World Wide Web (WWW 2007)*, Banff, Canada: ACM, May 2007, 601–610.

[123] M. Ter Louw and V. Venkatakrishnan, "Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers," in *30th Symposium on Security and Privacy (S&P 2009)*, Oakland, CA: IEEE, Jun. 2009, 331–346.

[124] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: Server-Side Detection of Cross-Site Scripting Attacks," in *24th Computer Security Applications Conference (ACSAC 2008)*, Anaheim, CA: IEEE, Dec. 2008, 335–344.

[125] P. Bisht and V. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks," in *5th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA 2008)*, Paris, France, Jul. 2008, 23–43.

[126] R. Antonello, S. Fernandes, C. Kamienski, D. Sadok, J. Kelner, I. GóDor, G. Szabó, and T. Westholm, "Deep Packet Inspection Tools and Techniques in Commodity Platforms: Challenges and Trends," *Journal of Network and Computer Applications*, vol. 35, no. 6, 1863–1878, Nov. 2012.

[127] F. Sabahi and A. Movaghar, "Intrusion Detection: A Survey," in *3rd International Conference on Systems and Networks Communications (ICSNC 2008)*, Sliema, Malta: IEEE, Oct. 2008, pp. 23–26.

[128] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection: A Survey," *ACM Computing Surveys*, vol. 41, no. 3, Jul. 2009.

[129] M. V. Mahoney and P. K. Chan, "PHAD: Packet Header Anomaly Detection for Identifying Hostile Network Traffic," Florida Tech., Technical Report CS-2001-4, 2001.

[130] M. V. Mahoney, "Network Traffic Anomaly Detection Based on Packet Bytes," in *ACM Symposium on Applied Computing (SAC 2003)*, Melbourne, FL: ACM, Mar. 2003, pp. 346–350.

[131] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications," in *10th International Symposium on Recent Advances in Intrusion Detection (RAID 2007)*, Queensland, Australia: Springer, Sep. 2007, pp. 63–86.

[132] D. Anderson, T. F. Lunt, H. Javitz, A. Tamaru, and A. Valdes, "Detecting Unusual Program Behavior Using the Statistical Components of NIDES," SRI International, Tech. Rep. SRI-CSL-95-06, May 1995.

[133] W. Lu, M. Tavallaee, and A. A. Ghorbani, "Detecting Network Anomalies Using Different Wavelet Basis Functions," in *6th Annual Conference on Communication Networks and Services Research (CNSR 2008)*, Halifax, Canada: IEEE, May 2008, pp. 149–156.

[134] A. Le, E. Al-Shaer, and R. Boutaba, "On Optimizing Load Balancing of Intrusion Detection and Prevention Systems," in *IEEE INFOCOM Workshops 2008*, Phoenix, AZ: IEEE, Apr. 2008.

[135] V. Sekar, R. Krishnaswamy, A. Gupta, and M. K. Reiter, "Network-wide Deployment of Intrusion Detection and Prevention Systems," in *6th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT 2010)*, Philadelphia, PA: ACM, Nov. 2010.

[136] M. Toulouse, B. Q. Minh, and P. Curtis, "A Consensus Based Network Intrusion Detection System," in *5th International Conference on IT Convergence and Security (ICITCS 2015)*, Kuala Lumpur, Malaysia: IEEE, Aug. 2015.

[137] R. P. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. R. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogrod, R. K. Cunningham, et al., "Evaluating Intrusion Detection Systems: The 1998 DARPA Off-Line Intrusion Detection Evaluation," in *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, vol. 2, Los Alamitos, CA: IEEE, Jan. 2000, 12–26.

[138] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian, "Internet Inter-Domain Traffic," in *ACM SIGCOMM 2010*, New Delhi, India: ACM, Aug. 2010, pp. 75–86.

[139] B. Ager, N. Chatzis, A. Feldmann, N. Sarrar, S. Uhlig, and W. Willinger, "Anatomy of a Large European IXP," in *ACM SIGCOMM 2012*, Helsinki, Finland: ACM, Aug. 2012, pp. 163–174.

[140] P. Richter, N. Chatzis, G. Smaragdakis, A. Feldmann, and W. Willinger, "Distilling the Internet's Application Mix from Packet-Sampled Traffic," in *Passive and Active Measurement Conference (PAM 2015)*, New York City, NY: Springer, Mar. 2015.

[141] L. Chappell, Wireshark Network Analysis The Official Wireshark Network Analyst Study Guide. Laura Chappell University, Mar. 2010.

[142] D. Guo, G. Liao, L. N. Bhuyan, B. Liu, and J. J. Ding, "A Scalable Multithreaded L7-filter Design for Multi-core Servers," in *4th ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS 2008)*, San Jose, CA: ACM, Nov. 2008, pp. 60–68.

[143] K. Namjoshi and G. Narlikar, "Robust and Fast Pattern Matching for Intrusion Detection," in *29th IEEE Conference on Computer Communications (INFOCOM 2010)*, San Diego, CA: IEEE, Mar. 2010.

[144] L. Deri, "nProbe: An Open Source NetFlow Probe for Gigabit Networks," in *TERENA Networking Conference (TNC 2003)*, Zagreb, Croatia, May 2003.

[145] C. Inacio and B. Trammell, "YAF: Yet Another Flowmeter," in *24th USENIX Conference on System Administration (LISA 2010)*, San Jose, CA, Nov. 2010.

[146] L. Deri, A. Cardigliano, and F. Fusco, "10 Gbit Line Rate Packet-To-Disk Using n2disk," in *32nd IEEE Conference on Computer Communications (INFOCOM 2013)*, Turin, Italy: IEEE, Apr. 2013, 3399–3404.

[147] S. Dharmapurikar and V. Paxson, "Robust TCP Stream Reassembly in the Presence of Adversaries," in *14th USENIX Security Symposium*, Baltimore, MD, Jul. 2005.

[148] P. Velan, T. Jirsik, and P. Celeda, "Design and Evaluation of HTTP Protocol Parsers for IPFIX Measurement," in *Advances in Communication Networking*, ser. Lecture Notes in Computer Science, T. Bauschert, Ed., vol. 8115, Springer Berlin Heidelberg, 2013, pp. 136–147.

[149] K. Salah and A. Kahtani, "Performance Evaluation Comparison of Snort NIDS Under Linux and Windows Server," *Journal of Network and Computer Applications*, vol. 33, no. 1, 6–15, Jan. 2010.

[150] R. Steward, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad, "Stream Control Transmission Protocol (SCTP) - Partial Reliability Extension," IETF, RFC 3758, May 2004.

[151] M. Primorac, E. Bugnion, and K. Argyraki, "How to Measure the Killer Microsecond," in *Workshop on Kernel-Bypass Networks (KBNets 2017)*, Los Angeles, CA: ACM, Aug. 2017, 37–42.

[152] F. Massicotte and Y. Labiche, "An Analysis of Signature Overlaps in Intrusion Detection Systems," in *41st International Conference on Dependable Systems & Networks (DSN 2011)*, Hong Kong, China: IEEE, Jun. 2011, 109–120.

[153] E. B. Lennon, "Testing Intrusion Detection Systems," National Institute of Standards and Technology, Information Technology Laboratory Bulletin Jul2003, Jul. 2003.

[154] A. Milenkoski, M. Vieira, S. Kounev, A. Avritzer, and B. D. Payne, "Evaluating Computer Intrusion Detection Systems: A Survey of Common Practices," *ACM Computing Surveys*, vol. 48, no. 1, Sep. 2015.

[155] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, "Mawilab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking," in *6th International Conference on emerging Networking Experiments and Technologies (CoNext 2010)*, Philadelphia, PA: ACM, Nov. 2010.

[156] N. Moustafa and J. Slay, "The Evaluation of Network Anomaly Detection Systems: Statistical Analysis of the UNSW-NB15 Data Set and the Comparison with the KDD99 Data Set," *ACM Information Security Journal: A Global Perspective*, vol. 25, no. 1-3, 18–31, Jan. 2016.

[157] J. McHugh, "Testing Intrusion DetectionSystems: A Critique Of The 1998 And 1999 Darpa Intrusion Detection System Evaluations As Performed By Lincoln Laboratory," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 4, 262–294, Nov. 2000.

[158] W. Bul'ajoul, A. James, and M. Pannu, "Improving Network Intrusion Detection System Performance through Quality of Service Configuration and Parallel Technology," *Elsevier Journal of Computer and System Sciences*, vol. 81, no. 6, pp. 981–999, Sep. 2015.

[159] T. Lukaseder, J. Fiedler, and F. Kargl, "Performance Evaluation in High-Speed Networks by the Example of Intrusion Detection Systems," in *11. DFN-Forum Kommunikationstechnologien*, P. Müller, B. Neumair, H. Reiser, and G. Dreo Rodosek, Eds., Bonn, Germany: Gesellschaft für Informatik e.V., Jun. 2018, pp. 33–42.

[160] S. Gallenmueller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of Frameworks for High-Performance Packet IO," in *11th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '15)*, Oakland, CA: IEEE, May 2015, 29–38.

[161] A. Branitskiy and I. Kotenko, "Network Attack Detection Based on Combination of Neural, Immune and Neuro-Fuzzy Classifiers," in *18th International Conference on Computational Science and Engineering (CSE 2015)*, Porto, Portugal: IEEE, Oct. 2015, 152–159.

[162] S.-J. Horng, M.-Y. Su, Y.-H. Chen, T.-W. Kao, R.-J. Chen, J.-L. Lai, and C. D. Perkasa, "A Novel Intrusion Detection System Based on Hierarchical Clustering and Support Vector Machines," *Elsevier Expert Systems with Applications*, vol. 38, no. 1, 306–313, Oct. 2011.

[163] M. Ektefa, S. Memar, F. Sidi, and L. S. Affendey, "Intrusion Detection Using Data Mining Techniques," in *International Conference on Information Retrieval & Knowledge Management (CAMP 2010)*, Selangor, Malaysia: IEEE, Mar. 2010, 200–203.

[164] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization," in *4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, Funchal, Portugal: INSTICC, Jan. 2018, pp. 108–116.

[165] B. Sangster, T. O'Connor, T. Cook, R. Fanelli, E. Dean, C. Morrell, and G. J. Conti, "Toward Instrumenting Network Warfare Competitions to Generate Labeled Datasets," in *2nd Workshop on Cybersecurity and Test (CSET 2009)*, Montreal, Canada: Usenix, Aug. 2009.

[166] K. Nasr, A. Abou-El Kalam, and C. Fraboul, "Performance Analysis of Wireless Intrusion Detection Systems," in *5th International Conference on Internet and Distributed Computing Systems (IDCS 2012)*, Fujian, China: Springer, Nov. 2012, 238–252.

[167] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A Scriptable High-Speed Packet Generator," in *15th Internet Measurement Conference (IMC 2015)*, Tokyo, Japan: ACM, Oct. 2015, 275–287.

[168] S. Patton, W. Yurcik, and D. Doss, "An Achilles' Heel in Signature-Based IDS: Squealing False Positives in SNORT," in *4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, Davis, CA: Springer, Oct. 2001.

[169] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," IETF, RFC 3986, Jan. 2005.

Acknowledgments

First of all I have to thank my parents, who encouraged me to attend university after I had already worked in IT for a couple of years. None of us would have expected in the beginning, that my studies would culminate in this thesis.

Further, I would definitely not have continued with a PhD if the atmosphere in our research group would not have been as friendly and helpful as it is. For this I greatly thank Falko and every single member of the research group in Innsbruck and in Paderborn.

I also have to thank all of my climbing and mountaineering friends, especially Franz and Sori, for giving me the breaks and adventures that I needed to flush my mind.

Finally and most importantly, I can not thank my partner Thekla enough for standing by me, supporting me and believing in me in these partially very stressful times.

Thanks again to all of you!