Flow-based Compromise Detection

Rick Hofstede

Graduation Committee

Chairman:	Prof. dr. P.M.G. Apers	
Promotor:	Prof. dr. ir. A. Pras	
Co-promotor:	Prof. Dr. rer. nat. G. Dreo Rodosek	

Members:

dr.	A. Sperotto	University of Twente, The Netherlands
Prof. dr.	P.H. Hartel	University of Twente, The Netherlands Delft University of Technology, The Netherlands TNO Cyber Security Lab, The Netherlands
Prof. dr. ir.	L.J.M. Nieuwenhuis	University of Twente, The Netherlands
Prof. dr. ir.	C.T.A.M. de Laat	University of Amsterdam, The Netherlands
Prof. dr. ir.	H.J. Bos	VU University, The Netherlands
Prof. Dr. rer. nat.	U. Lechner	Universität der Bundeswehr München, Germany
Prof. Dr. rer. nat.	W. Hommel	Universität der Bundeswehr München, Germany

Funding sources

EU FP7 UniverSelf – #257513
EU FP7 FLAMINGO Network of Excellence – #318488
EU FP7 SALUS – #313296
EIT ICT Labs – #13132 (Smart Networks at the Edge)
SURFnet GigaPort3 project for Next-Generation Networks

CTIT Ph.D. thesis series no. 16-384
Centre for Telematics and Information Technology
P.O. Box 217
7500 AE Enschede, The Netherlands

ISBN: 978-90-365-4066-7
ISSN: 1381-3617
DOI: 10.3990/1.9789036540667
http://dx.doi.org/10.3990/1.9789036540667

Typeset with LaTeX. Printed by Gildeprint, The Netherlands. Cover design by David Young.

FLOW-BASED COMPROMISE DETECTION

THESIS

to obtain
the degree of doctor at the University of Twente,
on the authority of the Rector Magnificus,
prof. dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Wednesday, June 29, 2016 at 16:45

by

Richard Johannes Hofstede

born on May 28, 1988
in Ulm, Germany.

This thesis has been approved by:

Prof. dr. ir. A. Pras (promotor)
Prof. Dr. rer. nat. G. Dreo Rodosek (co-promotor)

Acknowledgements

The list of people that contributed in one way or another to this thesis is almost endless. To avoid that I forget to include anyone here in this acknowledgement, I simply want to thank anyone who worked with me during my career as a Ph.D. student. You have helped shaping this thesis to what it has become. Thank you!

Rick Hofstede

Diepenveen, May 29, 2016

Abstract

Brute-force attacks are omnipresent and manyfold on the Internet, and aim at compromising user accounts by issuing large numbers of authentication attempts on applications and daemons. Widespread targets of such attacks are Secure SHell (SSH) and Web applications, for example. The impact of brute-force attacks and compromises resulting thereof is often severe: Once compromised, attackers gain access to remote machines, allowing those machines to be misused for all sorts of criminal activities, such as sharing illegal content and participating in Distributed Denial of Service (DDoS) attacks.

While the number of brute-force attacks is ever-increasing, we have seen that only few brute-force attacks actually result in a compromise. Those compromised devices are however those that require attention by security teams, as they may be misused for all sorts of malicious activities. We therefore propose a new paradigm in this thesis for monitoring network security incidents: *compromise detection*. Compromise detection allows security teams to focus on what is really important, namely detecting those hosts that have been compromised instead of all hosts that have been attacked. Speaking metaphorically, one could say that we target scored goals, instead of just shots on goals.

A straightforward approach for compromise detection would be host-based, by analyzing network traffic and log files on individual hosts. Although this typically yields high detection accuracies, it is infeasible in large networks; These networks may comprise thousands of hosts, controlled by many persons, on which agents need to be installed. In addition, host-based approaches lack a global attack view, i.e., which hosts in the same network have been contacted by the same attacker. We therefore take a network-based approach, where sensors are deployed at strategic observation points in the network. The traditional approach would be packet-based, but both high link speeds and high data rates make the deployment of packet-based approaches rather expensive. In addition, the fact that more and more traffic is encrypted renders the analysis of full packets useless. Flow-based approaches, however, aggregate individual packets into flows, providing major advantages in terms of scalability and deployment.

The main contribution of this thesis is to prove that flow-based compromise detection is viable. Our approach consists of several steps. First, we select two target applications, Web applications and SSH, which we found to be important targets of attacks on the Internet because of the high impact of a compromise and their wide deployment. Second, we analyze protocol behavior, attack tools and attack traffic to better understand the nature of these attacks. Third, we

develop software for validating our algorithms and approach. Besides using this software for our own validations (i.e., in which we use log files as ground-truth), our open-source Intrusion Detection System (IDS) *SSHCure* is extensively used by other parties, allowing us to validate our approach on a much broader basis. Our evaluations, performed on Internet traffic, have shown that we can achieve detection accuracies between 84% and 100%, depending on the protocol used by the target application, quality of the dataset, and the type of the monitored network. Also, the wide deployment of *SSHCure*, as well as other prototype deployments in real networks, have shown that our algorithms can actually be used in production deployments. As such, we conclude that flow-based compromise detection is viable on the Internet.

Contents

Introduction

The Internet has become a critical infrastructure that facilitates most digital communications in our daily lives, such as banking traffic, secure communications and video calls. This makes the Internet a prime attack target for criminals, nation-states and terrorists, for example [61]. When it comes to attacks, we there are two different classes that regularly make it to the news, namely those that are volumetric and aim at overloading networks and systems, such as Distributed Denial of Service (DDoS) attacks, and those that are particularly sophisticated, such as Advanced Persistent Threats (APTs). This is also confirmed by [158], but interestingly enough, it also reports brute-force attacks to be among the Top-3 of network attacks on the Internet [158], as shown in Figure 1.1. Although these attacks exist already for years, their popularity is increasing evermore [145], [147]. One of the main targeted services of these attacks is Secure SHell (SSH), a powerful protocol that allows for controlling systems remotely. The compromise of a target immediately results in adversaries gaining unprivileged control and the impact of a compromise is therefore remarkably high. With almost 26 million connected and scannable SSH daemons in November 2015 according to Shodan,[1] SSH daemons are a popular and widely available attack target [12].

[1] `https://www.shodan.io`

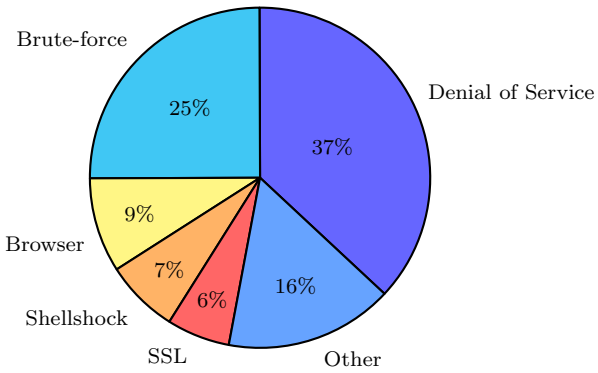


Figure 1.1: Top network attacks in 2015, from [158].

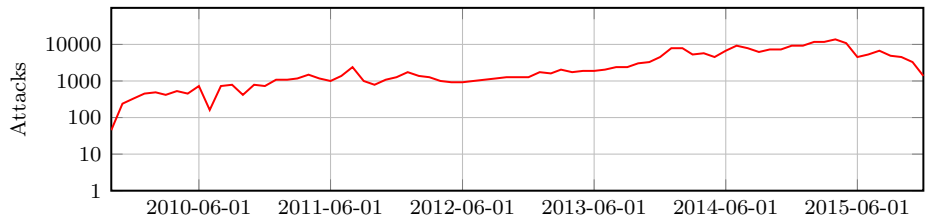


Figure 1.2: SSH brute-force attacks observed by OpenBL, from [133].

The threat of SSH attacks was also stressed by the *Ponemon 2014 SSH Security Vulnerability Report*: 51% of the surveyed companies had been compromised via SSH in the last 24 months [160]. These compromises can be accounted mostly to poor key management, causing former employees to still have access to enterprise systems after they left the company. It is however also generally known that compromises can be the result of brute-force attacks. In campus networks, such as the network of the University of Twente (UT) with roughly 25,000 active hosts, we observe approximately 115 brute-force attacks per day, while in backbone networks, such as the Czech National Research and Education Network (NREN) CESNET, it is not uncommon to observe more than 700 per day. Even more attacks may be expected in the future; Several renowned organizations, such as OpenBL[2] and DShield,[3] report a tripled number of SSH attacks between August 2013 and April 2014. In Figure 1.2, we show the number of observed SSH brute-force attacks against sensors deployed by OpenBL worldwide, which underlines the rapid increase in popularity of these attacks. In April 2015, the threat intelligence organization Talos, together with Tier 1 network operator Level 3 Communications, stopped SSH brute-force attacks of a group named *SSHPsychos* or *Group 93*, which generated more than 35% of the global SSH network traffic [157]. Since no legitimate traffic was found to be originating from the attacking networks, those networks were simply disconnected from the Internet. Note that this drop is also visible in Figure 1.2.

Besides SSH compromises that typically have a high impact, there is another class of hacking targets on the Internet that receives lots of attention in brute-force attacks: Web applications in general and Content Management Systems (CMSs) like Wordpress, Joomla and Drupal in particular [130]. Web applications and the aforementioned CMSs are characterized by a very large number of deployments, as they are used for powering roughly 30% of all Web sites on the Internet, with Wordpress being the dominant solution with a market share of 25% [153]. The security company Sucuri visualizes failed login attempts on Wordpress instances behind their protection services, which shows an increase month after month, up to a factor eight over six months in 2015 [147], as shown

[2] `http://www.openbl.org`
[3] `http://www.dshield.org`

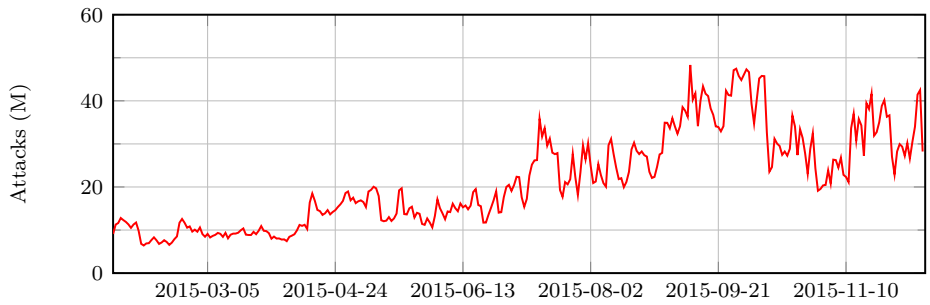


Figure 1.3: Brute-force attacks against Wordpress instances behind Sucuri's protection service, from [147].

in Figure 1.3. Another security company, Imperva, even acknowledged in November 2015 that CMSs are attacked three times more often than non-CMS Web applications (Wordpress even 3.5 times more often) and that Wordpress is targeted seven times more for SPAM and Remote File Inclusion[4] attacks than non-CMS applications [156]. The fact that anybody can use CMSs, even people with limited technical skills that are unaware of security threats and measures, makes CMSs a prime attack target, especially with regard to the following aspects:

- **Vulnerabilities** – Because of the interaction between Web browser and Web server that is needed for editing remote content, major parts of CMSs are built using code that is executed dynamically. In contrast to static code, such as pure HyperText Markup Language (HTML) pages, dynamic code (e.g., PHP) is executed by the Web server. As such, once an attacker is able to modify the code, arbitrary commands can be executed and modified content served to clients. Although patches for the aforementioned CMSs are released periodically, talks with Dutch Top 10 Web hosting companies have revealed that approximately 80% of all CMS instances runs on outdated software.

- **Weak passwords** – Although it is often advocated to use unique and random passwords, one per site or instance, people tend to use memorable passwords. Since memorable passwords limit the level of security that password authentication can possess [82], weak passwords form a major security risk. Even though this is true for any service or application that is exposed to the Internet, the fact that CMSs are designed to be used by people with limited technical skills worsens this aspect.

[4]Remote File Inclusion (RFI) attacks exploit poorly crafted 'dynamic file inclusion' mechanisms, such that arbitrary code can be loaded into and executed by Web applications.

CMSs have received an increasing amount of negative attention because of attacks, vulnerabilities and compromises in recent years. Also the type of attacks has changed over time. For example, reports initially described large increases in the number of observed brute-force attacks against CMSs and compromised CMSs participating in botnets ([132], [144], [152]). In later years, however, the focus shifted towards misusing (compromised) blogs for amplification in DDoS attacks ([111], [150]). The fact that CMSs are so widespread make that misuse can result in a new dimension of attacks, and that any vulnerability or weakness can be exploited in great extent.

It is clear that attacks against SSH daemons and Web applications are omnipresent and manyfold. Detecting all these attacks and acting upon them is not only a resource-intensive task, but it requires a new paradigm for handling them. While brute-force attacks may result in severe damage, only few of them are actually successful in the sense that they result in a compromise. Compromised devices are the dangerous ones that require attention, as they may be misused for all sorts of malicious activities. We therefore target these in our novel paradigm for monitoring networks, which we refer to as *compromise detection*. Our validations (presented in Chapter 4 and 5) have shown that we observe only a handful of compromises in thousands of incidents per month. As such, we conclude that compromise detection provides much more precise information on compromised devices and weaknesses than regular attack detection (i.e., a device has been compromised vs. a device was attacked), and results in a major scalability gain and complexity reduction in terms of incident handling.

1.1 Compromise Detection

Compromised devices are the core building blocks of illegal activities on the Internet. Once compromised, they can be used for sending SPAM messages [122], [143], launching DDoS attacks [111], distributing illegal content [122] and joining botnets [132], just to name some examples. Reasons for using compromised devices for such activities are manyfold, such as impersonation to hide illegal activities, and better technical infrastructure (e.g., network bandwidth) to be able to perform more illegal activities or reach a wider audience.

Many attacks and incident reports do not require immediate action. For example, in the case of SSH, our measurements and validations have shown that in a campus subnetwork with 80 workstations and servers, zero or only a few compromises occur per month, while more than 10,000 attacks were observed in total. Similar proportions were observed when validating attacks against Web applications in the network of a large Web hosting provider: Also in the course of one month, only one compromise was observed, out of almost 800 attacks. The observed ratios between attacks and compromises likely stem from the nature of brute-force attacks; Attack tools often use *dictionaries*, lists of frequently-used passwords, so if Web applications are protected using randomly generated

	Host-based detection	Network-based detection
Accuracy	+	–
Scalability	–	+
View	–	+

Table 1.1: Host-based vs. network-based detection.

passwords, for example, brute-force attacks are much less likely to succeed. For this reason, detecting compromises rather than just attacks is an important step forward for security teams that are overloaded by attacks and incident reports – a situation that is the rule rather than the exception, as confirmed by the 2015 Black Hat Europe conference's attendee survey [124].

In the remainder of this section, we address two types of compromise detection: host-based and network-based. The difference is the observation point where data is collected for performing compromise detection: on end hosts (Section 1.1.1) or at central observation points in the network (Section 1.1.2). The key characteristics of both types are discussed in the following subsections and summarized in Table 1.1.

1.1.1 Host-based Compromise Detection

Compromise detection is traditionally performed in a host-based fashion, i.e., by running Intrusion Detection Systems (IDSs) on end systems like servers and workstations. As such, IDSs have access to network interfaces and file systems, allowing them to achieve high detection rates with few false positives and negatives. Due to the availability of fine-grained information, such as network traffic and log files, they are able to detect various phases and aspects of attacks, ranging from simple port scans to compromises. A fundamental problem of host-based approaches is their poor scalability; In environments where IT service departments and security officers do not have access to every machine, it is infeasible to install and manage detection software on every device. This can be exemplified again using the campus network of the UT, with 25,000 active hosts; Controlling all these hosts would require an extensive infrastructure and paradigms like Bring Your Own Device (BYOD) will always yield unsecured devices. For this reason, network-based approaches may be used for monitoring networked systems, where information is gathered at central observation points in the network rather than on end systems. Another problem of host-based approaches is their isolated view on attacks; Since information can only be gathered about the system on which it is deployed, a broader view on the attack is missing, e.g., whether multiple hosts are targeted by the same attacker at the same time. Some works, such as [55], have worked around this issue by developing a system for information sharing between individual hosts.

1.1.2 Network-based Compromise Detection

Network-based IDSs are far behind when it comes to compromise detection, as they generally report on the *presence* of attacks (e.g., [28]), regardless of whether an attack was successful or not. This is mostly because of the coarser-grained information that is available to network-based solutions; To be able to cope with high link speeds and large amounts of network traffic, information is typically collected and analyzed in aggregated form, which means that details are lost by definition. In this thesis, we investigate how compromise detection can be performed in a network-based fashion, to overcome the limitations of a host-based approach: isolated views on attacks as a consequence of host-based observation points, as well as limited scalability. More precisely, we aim for a network-based approach that can be deployed at central observation points in the network such that they have a global view on attacks, without the need to have control over every monitored device in the network.

To perform network-based compromise detection, it is crucial to have a powerful network monitoring system in place. In the next section, we discuss various approaches for performing network monitoring.

1.2 Network Monitoring

Network monitoring systems can generally be classified into two categories: active and passive. Active approaches, such as implemented by tools like Ping and Traceroute, as well as management protocols like Simple Network Management Protocol (SNMP), inject traffic into a network to perform different types of measurements. Passive approaches observe existing traffic as it passes by an observation point and therefore observe traffic generated by users. One passive monitoring approach is packet capture. This method generally provides most insight into the network traffic, as complete packets can be captured and further analyzed. In situations where it is unknown at the moment of capturing which portion of the packet is relevant or if parts of packet payloads need to be analyzed, packet capture is the method of choice. Also, when meta-data like packet inter-arrival times are needed, packet capture provides the necessary insights. However, the fact that more and more traffic is encrypted diminishes packet capture. Also, in high-speed networks with line rates of 100 Gbps and beyond, packet capture requires expensive hardware and substantial infrastructure for storage and analysis.

Another passive network monitoring approach that is more scalable for use in high-speed networks is flow export, in which packets are aggregated into flows and exported for storage and analysis. Initial works on flow export date back to the nineties and became the basis for modern protocols, such as NetFlow and IP Flow Information Export (IPFIX) [2]. Although every flow export protocol may use its own definition of what is considered a flow, a flow is generally defined for

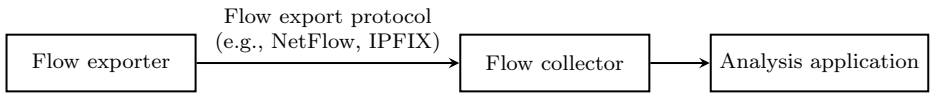


Figure 1.4: High-level overview of a flow monitoring setup.

(NetFlow and) IPFIX in Request for Comment (RFC) 7011 as "a set of IP packets passing an observation point in the network during a certain time interval, such that all packets belonging to a particular flow have a set of common properties". These common properties may include packet header fields, such as source and destination IP addresses and port numbers, interpreted information based on packet contents, and meta-information.

Data exported using flow export technologies has seen a huge uptake since communication providers are enforced to retain connection meta-data for several months or years. This is because the retainment policies prescribe exactly the information that is typically provided in flow data. While the typical data retention laws are in place for years already, traffic meta-data is still widely discussed in politics. For example, the Dutch government has passed a bill only in 2015 that introduces an obligation for so-called 'data controllers' to notify the Dutch Data Protection Authority of data security breaches, effective as of January 1, 2016 [126]. This bill forces data controllers to record traffic meta-data, again, using flow data. So in short, flow data has been in place for a long time already and is still very relevant in all sorts of telecommunication legislation.

A high-level overview of flow monitoring setups is shown in Figure 1.4. The figure shows a *flow exporter* that receives packets, aggregates them into flows and exports flow data to a *flow collector* for storage and preliminary analysis. A flow export protocol, such as NetFlow or IPFIX, is used for transmitting the flow data to the flow collector. Once the data is recorded by the flow collector, *analysis applications* may be used for analyzing the data. An example of how parts of Internet Protocol (IP) packets are used in flow records is shown in Figure 1.5.

In addition to their suitability for use in high-speed networks, flow export protocols and technologies provide several other advantages above regular packet capture.

1. Flow export is frequently used to comply to data retention laws. For example, communication providers in Europe are enforced to retain connection data, such as provided by flow export, for a period of between six months and two years "for the purpose of the investigation, detection and prosecution of serious crime" [13], [125].

2. Flow export protocols and technologies are widely deployed, mainly due to their integration into high-end packet forwarding devices, such as routers, switches and firewalls. For example, a recent survey among both commercial and research network operators has shown that 70% of the participants

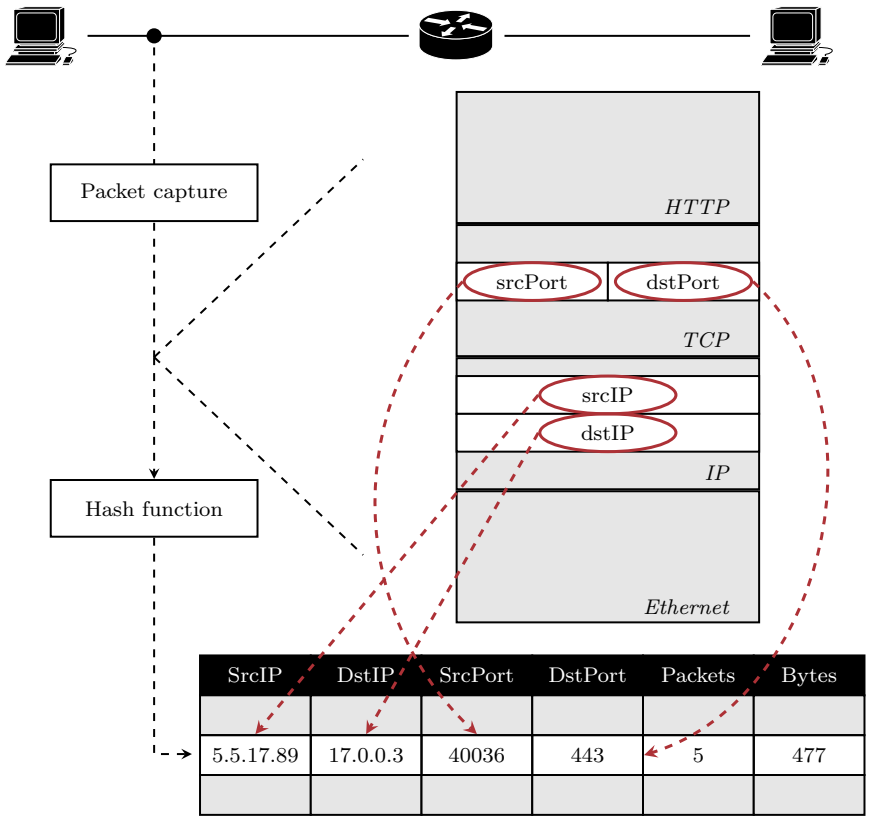


Figure 1.5: Packet header fields (and content) exported in flow data.

have devices that support flow export [70]. As such, no additional capturing devices are needed, which makes flow monitoring less costly than regular packet capture.

3. Flow export is well understood, since it is widely used for security analysis, capacity planning, accounting, and profiling, among others.

4. Significant data reduction can be achieved – in the order of 1/2000 of the original volume, as shown in Chapter 2 – since packets are aggregated after they have been captured.

5. Flow export is usually less privacy-sensitive than packet export, since traditionally only packet headers are considered and payloads not even captured. However, since researchers, vendors and standardization bodies are working on the inclusion of application information in flow data, the advantage of performing flow export in terms of privacy is fading.

6. Since flow export was designed to operate mostly on packet header fields, it is neither hindered by any application-layer encryption, such as Transport Layer Security (TLS) and recent protocol developments like Google QUIC that aim at multiplexing streams over User Datagram Protocol (UDP).[5]

Although flow export provides many advantages over packet-based traffic analysis, it is also subject to several deficiencies that are inherent to the design of the respective protocols and technologies:

1. The aggregated nature and therefore coarser granularity of flow data – in many respects a major advantage – makes certain types of analysis more challenging. For example, while retransmitted packets, which are a sign of connectivity issues, can easily be identified in a set of packets, they cannot be discriminated in regular flow data.

2. The advantages provided by flow export usually excuse the coarser data granularity, as long as the flow data reflects the actual network traffic precisely. However, the flow export process may introduce artifacts in the exported data, i.e., inaccuracies or errors, which may impair flow data analyses.

3. Flow monitoring systems are particularly susceptible to network flooding attacks. This is because flow records are meant to resemble connections, but if every connection consists of only one or two packets, the scalability advantage that was achieved by means of aggregation is lost. Moreover, depending on the nature of the attack, the attack traffic might even be amplified by the overhead of (a) every flow record and (b) export protocols like NetFlow and IPFIX. It is therefore important that flow exporters are resilient against attacks to avoid 'collapses' under overload, effectively 'blinding' the monitoring infrastructure.

Flow data is a proven source of information for detecting various types of security incidents, such as DDoS attacks and network scans [20]. However, research and literature based on flow data analysis does hardly deal with the identified deficiencies of flow export. This is because validations are often done in lab environments or local area networks, where flow export devices operate under ideal conditions.

1.3 Objective, Research Questions & Approach

1.3.1 Objective

In the previous section, we have made a case for compromise detection, a novel paradigm that makes security analysts focus on what is really important: actual

[5] `https://www.chromium.org/quic`

compromises. Or, in analogical terms: We are not so much interested in just shots, but in scored goals. To perform compromise detection, several network-based approaches can be used and we have shown that flow monitoring provides several advantages over packet-based alternatives. Disadvantages can however be identified as well, which need to be minimized and preferably overcome. The objective of this thesis can therefore be formulated as follows:

> *Investigate how compromise detection can be performed on the Internet using flow monitoring technology.*

In other words, the objective of this thesis is to investigate whether compromise detection is feasible on the Internet at all, or whether it still is at an academic stage that allows primarily for lab deployment. We explicitly say *on the Internet* to make clear that we do not target lab environments, but (large) networks that are in daily production usage.

1.3.2 Research Questions & Approach

In light of the objective of this thesis, a first and elementary item to address is how to perform sound flow monitoring. Flow monitoring is used as the means for capturing network traffic for use in our compromise detection paradigm, as it provides a scalable and aggregated means for analyzing traffic in large and high-speed networks. Even though flow monitoring is widely deployed and typically well understood, there is a considerable number of pitfalls that hinder reliable operation and impair flow data analysis. For example, many flow exporters do not strictly adhere to standards and specifications, and configurations are often not considered as precisely as one would expect. In this thesis, we investigate all the various stages present in flow monitoring setups, as well as common pitfalls, such that we obtain a solid basis for performing *sound* flow measurements. In this context, flow measurements are considered sound if the exported flow data reflects the original network traffic precisely. We summarize our first research question as follows:

> **RQ1** – *Can flow monitoring technology be used for compromise detection?*

Our approach for answering this research question consists of multiple steps. We start with surveying literature where relevant and applicable. Complementary to this survey, we include information based on our own experience. This experience has been gained in a variety of ways: research in the area of flow export, involvement in the standardization of IPFIX, operational experience by working for a leading company in the area of flow-based malware detection, talks with network operators, and experience in developing both hardware-based and software-based flow exporters. We also include measurements to illustrate and

provide more examples and insights into the presented concepts. Then, we continue with analyzing the exported flow data of a range of widely-used flow monitoring devices, to compare the flow data quality to specifications and configurations.

After investigating how to perform sound flow measurements, as well as related pitfalls, we can work towards the main contribution of this thesis: *flow-based compromise detection*. Early work in this area has been shortly addressed in another thesis ([81]). Based on the lessons learned of both [81] and RQ1, we target flow-based compromise detection in a comprehensive manner in this thesis, such that it may be used on the Internet. In this context, we define our second research question as follows:

> **RQ2** – *How viable is compromise detection for application on the Internet?*

To address this research question, we focus on two popular brute-force attack targets: SSH and Web applications. Compromises resulting thereof typically provide the attacker with system-level access that can be misused for various purposes. For both SSH and Web applications, which have been selected because of the large impact of a compromise and wide deployment, respectively, we take the following approach. First, we investigate the nature of the involved protocol, to understand the typical protocol messages and message sequences. Then, we harvest attack tools by operating honeypots, visiting hacker fora and analyzing code snippets on public code sharing Web sites, to learn about the techniques employed to compromise hosts. Based on the lessons learned, we develop detection algorithms and validate them as as follows:

- We compare our detection results with various ground-truth datasets, such as log files, and perform multiple large-scale validations that enable us to express the performance of our algorithms in terms of frequently-used evaluation metrics. The datasets have been collected on the campus network of the UT, over the course of one month, and consist of flow data and log files of almost 100 servers and workstations. It must be stressed here that the use of realistic datasets that have been collected in open networks is one of the cornerstones of our validation.

- We implement an open-source SSH intrusion detection system, *SSHCure*.[6] We developed *SSHCure* as a demonstrator for our compromise detection algorithms, which allowed us to obtain community feedback on detection results of our work in other networks, such as NRENs and other backbone networks.

[6] To be pronounced as *she-cure*.

In the context of this thesis, we consider compromise detection *viable* if detection accuracies higher than 80% are achieved, and the detection results of prototypes may be used in production.

The compromise detection algorithms and prototypes presented in this thesis work well as long as the flow dataset has been collected in a sound fashion. However, as explained in Section 1.2, by their nature to make flows resemble connections, flow monitoring systems are susceptible to attacks that consist of large numbers of connections, especially if those connections are very small in terms of packets and bytes. In those situations, the scalability gain of flow monitoring systems is lost, due to the overhead of flow accounting. This is a widely known problem that has been confirmed by both vendors of flow monitoring devices and large network operators. Exemplary attacks are very large network scans and flooding attacks, such as DDoS attacks. To investigate how to overcome this resilience problem of flow monitoring systems, we define our third and final research question as follows:

> **RQ3** – *Which components of flow monitoring systems are susceptible to flooding attacks, how can these attacks efficiently be detected and how can the resilience of flow monitoring systems be improved?*

We address this research question by taking the following approach. First, we investigate which metrics are significative for flooding attacks and how they can be retrieved from flow monitoring systems. Since flow data analysis is typically performed after storage and preliminary analysis on a flow collection device, we intuitively assume that moving detection closer to the data source, i.e., towards the flow export device, may enable us to filter out attack traffic before it reaches the monitoring infrastructure. As such, we develop a lightweight detection algorithm and implement it as part of a prototype that can be deployed on dedicated flow export devices. In light of a wide deployment and applicability of our approach, we even investigate the possibility of deploying our prototype on packet forwarding devices with flow export support. We validate our work in both the Czech NREN CESNET and the campus network of the UT, for a period of several weeks.

1.4 Contributions

The main contribution of this thesis is a new network attack detection paradigm – *compromise detection* – that targets *successful* attacks instead of *all* attacks. Compromise detection therefore reduces the number of attacks and incident reports to be handled by security teams drastically. We demonstrate that our new paradigm operates on aggregated network data, i.e., flow data. This flow data is widely available (in roughly 70% of all networks operated by 135 participants involved in a large survey in 2013 [70]) and greatly reduces the amount of data

to be analyzed by monitoring systems, due to the aggregation of individual packets into flows. We prove that flow-based compromise detection is viable on the Internet. Moreover, our IDS *SSHCure*, which includes our compromise detection algorithms for SSH, has evolved into a real open-source software project that serves a community, and is deployed in many networks around the world.

Besides our main contribution, we identify the following specific contributions:

- An analysis of artifacts in flow data from widely-used flow export devices (Chapter 3).
- Algorithms and prototypes for the detection of SSH compromises, validated on the Internet (Chapter 4).
- Algorithms and prototypes for the detection of Web application compromises, validated on the Internet (Chapter 5).
- Algorithms and prototypes for the detection of flooding attacks, like DDoS attacks, in an efficient manner using flow export devices and in such a way that the monitoring infrastructure is resilient against the attack (Chapter 6).
- Demonstration of how measuring Transmission Control Protocol (TCP) control information and retransmissions is beneficial for *any* flow data analysis (Section 4.4).
- Annotated datasets for the evaluation of flow-based IDSs, published as part of the traces repository on the SimpleWeb.[7]
- Open-source intrusion detection software *SSHCure*, the first flow-based IDS that could report on compromises. Over the last years, we have seen major interest in *SSHCure* from many parties, ranging from small network operators to nation Computer Security Incident Response Teams (CSIRTs).
- Prototype implementations and deployment results of IDSs on forwarding devices and flow probes, published as open-source software.

Besides these specific contributions, we have developed the first comprehensive tutorial on flow monitoring using NetFlow and IPFIX, covering the full spectrum from packet capture to data analysis (Chapter 2).

1.5 Thesis Organization

In light of this thesis' objective and its supporting research questions, we have organized this thesis in three parts, each addressing one research question. The remainder of this section summarizes each thesis part and the chapters they comprise. Additionally, we visualize the structure of this thesis in Figure 1.6.

[7] `http://www.simpleweb.org/wiki/SSH_datasets`

Part I – Generic Flow Monitoring
The first part of this thesis, consisting of Chapter 2 and 3, fully covers RQ1 and consists of a comprehensive tutorial on flow monitoring and a description of widely observed flow measurement artifacts. Although the content in this part of the thesis provides a solid basis for understanding the remainder of this thesis, it applies to basically any flow monitoring system and its application is therefore not limited to compromise detection alone.

Chapter 2 – Flow Measurements

Flow export technology can nowadays be found in many networking devices, mostly integrated in packet forwarding devices or as dedicated flow export appliances ('probes'). Although this technology is often presented as being *plug-and-play* and the exported flow data as 'universal', there are many pitfalls that may impair the flow data quality. We therefore investigate in this chapter, from a theoretical point of view, whether flow data is suitable for use in compromise detection. Then, in Chapter 3, we complement this investigation with a practical point of view, by analyzing flow data from a wide range of flow export devices.

Provided that flow monitoring systems are complex and feature many variables, understanding their components and knowing their pitfalls is key to performing sound measurements. However, there is no comprehensive tutorial available that explains all the ins and outs of flow monitoring. This chapter bridges this gap. We start with the history of flow export in the 80's and 90's and compare flow export by means of NetFlow and IPFIX

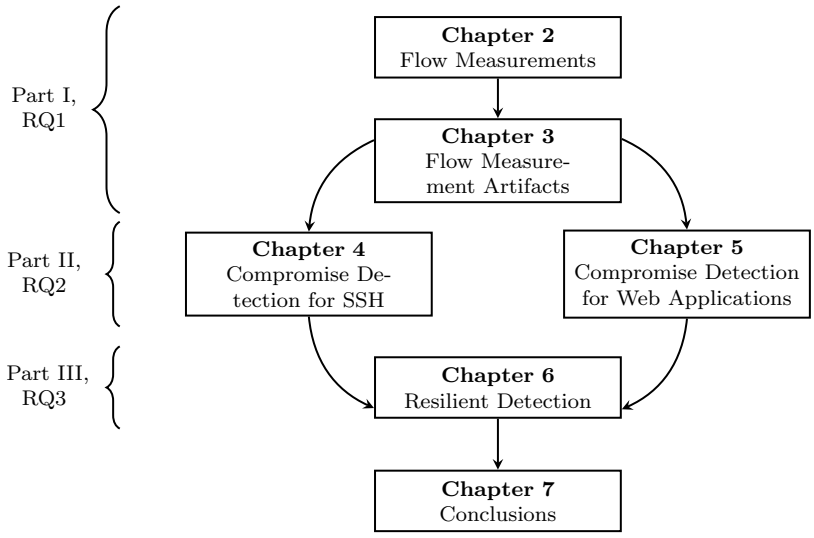


Figure 1.6: Thesis organization.

with other technologies with *flow* in the name, such as sFlow and OpenFlow, which do not solve exactly the same problems as flow export. Then, we show the core building blocks of any flow monitoring setup, from packet capture to data analysis, and describe each of them in subsequent sections. Besides an analysis of specifications and state-of-the-art, we add insights based on our own experience and complement this with measurements. The content of this chapter is published in:

- R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, A. Pras. *Flow Monitoring Explained: From Packet Capture to Data Analysis with NetFlow and IPFIX*. In: IEEE Communications Surveys & Tutorials, Vol. 16, No. 4, 2014

Chapter 3 – Flow Measurement Artifacts

Implementation decisions or errors, operating conditions or platform limitations may cause flow data to not resemble the original packet stream precisely. Inaccuracies or errors in flow data is what we refer to as flow data artifacts. Although artifacts do not necessarily impair the analysis of flow data, it is of utmost importance to at least be aware of their presence, e.g., to avoid interpretation errors. In this chapter, we demonstrate the omnipresence of artifacts by comparing the flow data of six different and widely deployed flow export devices. Our measurements show, for example, that the quality of flow data of dedicated flow export devices (probes) is superior in quality to flow data exported by packet forwarding devices, and that flow data from widely-deployed, high-end devices features plenty of artifacts. Based on our observations, we conclude whether compromise detection based on flow data is feasible in practice. The content of this chapter is published in:

- R. Hofstede, I. Drago, A. Sperotto, R. Sadre, A. Pras. *Measurement Artifacts in NetFlow Data*. In: Proceedings of the 14th International Conference on Passive and Active Measurement, PAM 2013, 18-19 March 2013, Hong Kong, China – **Best Paper Award**

Part II – Compromise Detection

The second part of this thesis, consisting of Chapter 4 and 5, covers the main contribution: *flow-based compromise detection*. We demonstrate the feasibility of our novel paradigm to detect compromises instead of attacks, and investigate compromise detection for two widespread targets of brute-force attacks: SSH daemons and Web applications. Both targets are discussed and validated in their own chapter.

Chapter 4 – Compromise Detection for SSH

In this chapter, we investigate whether and how well flow-based compromise detection can be performed for SSH. We start by describing related work

in this area and analyzing key characteristics of brute-force attacks against SSH daemons, resulting in a three-phase attack model that will be used throughout this thesis. This model helps in understanding why compromises are always preceded by a brute-force attack and how the identification of network scans may improve the detection of compromises. Based on this model, we present our basic detection approach that aims at identifying deviating connections in network traffic that could signify a compromise. This approach is rather generic and can in principle be used for detecting compromises over other protocols as well. Although it catches many attacks and is taken up by related works as well, we demonstrate how it is limited by network artifacts like TCP retransmissions and control information that cause compromises or even complete attacks to stay under the radar of IDSs. After that, we proceed in two directions to enhance the detection results of our approach. First, we investigate network artifacts in detail and enhance our measurement infrastructure such that we can overcome any deficiencies caused by artifacts. Second, we enhance our detection approach by including SSH-specific knowledge, which allows us to identify protocol behavior that would be marked as a compromise otherwise. We validate our work based on production traffic of more than 100 devices on the campus network of the UT. The content of this chapter is published in:

- L. Hellemons, L. Hendriks, R. Hofstede, A. Sperotto, R. Sadre, A. Pras. *SSHCure: A Flow-based SSH Intrusion Detection System.* In: Dependable Networks and Services. Proceedings of the 6th International Conference on Autonomous Infrastructure, Management and Security, AIMS 2012, 4-8 June 2012, Luxembourg, Luxembourg – **Best Paper Award**
- R. Hofstede, L. Hendriks, A. Sperotto, A. Pras. *SSH Compromise Detection using NetFlow/IPFIX.* In: ACM Computer Communication Review, Vol. 44, No. 5, 2014
- M. Jonker, R. Hofstede, A. Sperotto, A. Pras. *Unveiling Flat Traffic on the Internet: An SSH Attack Case Study.* In: 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015, May 11-15, 2015, Ottawa, Canada

Chapter 5 – Compromise Detection for Web Applications

We investigate in this chapter how to perform flow-based compromise detection for Web applications. In contrast to SSH, the detection of attacks against Web applications by means of flow data is an untouched area of research, meaning that there is hardly any related work to be consulted. We use our expertise in the area of SSH, including our three-phase attack model, to approach this challenge. To explore the area of Web application attacks, we start again by taking a signature-based approach to detect brute-force authentication attempts. Then, based on the lessons learned in

the area of SSH, we develop a new approach: Clustering methods allow us to group connections in an attack that are similar in terms of packets, bytes and duration. All these similar connections are likely to feature (failed) authentication attempts, while outliers may be the compromise that we aim to identify. To validate our work, we use network traffic of a large, Dutch Web hosting provider that was collected during one month in 2015, and consists of traffic towards more than 2500 Web applications. The content of this chapter is published in:

- O. van der Toorn, R. Hofstede, M. Jonker, A. Sperotto. *A First Look at HTTP(S) Intrusion Detection using NetFlow/IPFIX*. In: 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015, May 11-15, 2015, Ottawa, Canada
- R. Hofstede, M. Jonker, A. Sperotto, A. Pras. *Flow-based Web Application Brute-force Attack & Compromise Detection* (under review)

Part III – Resilient Detection
One of the deficiencies of flow monitoring systems is that they are susceptible to flooding (e.g., DDoS) attacks by nature. In the third and last part of this thesis, we address this deficiency.

Chapter 6 – Resilient Detection
To improve the resilience of flow monitoring systems against flooding attacks, we investigate in this chapter whether we can equip flow export devices with a lightweight module for detecting and ultimately filtering out attack traffic. This avoids that flooding traffic will reach data collection and analysis devices and ensures that monitoring devices are not blinded by attacks. Moreover, detection results of this approach may even be fed into a firewall to block the attack not only from reaching the monitoring infrastructure, but also from reaching regular networked devices. The detection algorithm employed 'learns' connection patterns of the observation point over time and recognizes sudden deviations. We validate our approach on dedicated flow export devices deployed in the network of a large European backbone network operator and on packet forwarding devices on the campus network of the UT. The content of this chapter is published in:

- R. Hofstede, V. Bartoš, A. Sperotto, R.Sadre, A. Pras. *Towards Real-Time Intrusion Detection for NetFlow and IPFIX*. In: Proceedings of the 9th International Conference on Network and Service Management, CNSM 2013, 15-17 October 2013, Zürich, Switserland
- D. van der Steeg, R. Hofstede, A. Sperotto, A. Pras. *Real-time DDoS Attack Detection for Cisco IOS using NetFlow*. In: 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015, May 11-15, 2015, Ottawa, Canada

Chapter 7 – Conclusions

In the last chapter of this thesis, we summarize our findings and contributions by answering our research questions. Also, we elaborate on various directions for future work.

Part I

Generic Flow Monitoring

CHAPTER 2

Flow Measurements

Many papers, specifications and other documents on NetFlow and IPFIX have been written over the years. They usually consider the proper operation of flow export protocols and technologies, as well as the correctness of the exported data, as a given. We have however seen that these assumptions often do not hold. We therefore investigate in Chapter 2 and 3 whether the flow export technologies NetFlow and IPFIX can be used for compromise detection. We start in this chapter by analyzing the individual components of flow monitoring setups, the interworking of these components, configuration options, and pitfalls. After that, in Chapter 3, we take a similar approach for investigating the practical suitability of flow export devices for compromise detection. The objective of this tutorial-style chapter is to provide a clear understanding of flow export and all stages in a typical flow monitoring setup, covering the complete spectrum from packet capture to data analysis. Based on our observations, we conclude whether compromise detection based on flow data is feasible in theory.

The paper related to this chapter is [11], which was published in IEEE Communications Surveys & Tutorials.

The organization of this chapter is as follows:

- *Section 2.1 describes the history of flow export and compares NetFlow and IPFIX to other related and seemingly-related protocols, such as sFlow and OpenFlow.*
- *Section 2.2 introduces the various stages that make up the architecture of flow monitoring setups.*
- *Sections 2.3–2.5 cover the first three stages of flow monitoring: Packet Observation, Flow Metering & Export, and Data Collection. The fourth and final stage, Data Analysis, is exemplified by Chapter 4 and 5.*
- *Section 2.6 outlines the most important lessons learned.*
- *Section 2.7 concludes this chapter.*

2.1 History & Context

In this section, we discuss both the history of flow monitoring and present flow monitoring in a broader context by comparing it to related technologies. The chronological order of the main historic events in this area is shown in Figure 2.1 and will be covered in Section 2.1.1. A comparison with related technologies and approaches is provided in Section 2.1.2.

2.1.1 History

The published origins of flow export date back to 1991, when the aggregation of packets into flows by means of packet header information was described in [88]. This was done as part of the Internet Accounting (IA) Working Group (WG) of the Internet Engineering Task Force (IETF). This WG concluded in 1993, mainly due to lack of vendor interest. Also the then-common belief that the Internet should be free, meaning that no traffic capturing should take place that could potentially lead to accounting, monitoring, etc., was a reason for concluding the WG. In 1995, interest in exporting flow data for traffic analysis was revived by [6], which presented a methodology for profiling traffic flows on the Internet based on packet aggregation. One year later, in 1996, the new IETF Real-time Traffic Flow Measurement (RTFM) WG was chartered with the objectives of investigating issues in traffic measurement data and devices, producing an improved traffic flow model, and developing an architecture for improved flow measurements. This WG revised the Internet Accounting architecture and, in 1999, published a generic framework for flow measurement, named RTFM Traffic Measurement System, with more flexibility in terms of flow definitions and support for bidirectional flows [89]. In late 2000, having completed its charter, the RTFM WG was concluded. Again, due to vendors' lack of interest, no flow export standard resulted.

In parallel to RTFM, Cisco worked on its flow export technology named NetFlow, which finds its origin in switching. In flow-based switching, flow information is maintained in a *flow cache* and forwarding decisions are only made in the control plane of a networking device for the first packet of a flow. Subsequent packets are then switched exclusively in the data plane [114]. The value of the information available in the flow cache was only a secondary discovery [119] and the next step to export this information proved to be relatively small. NetFlow was patented by Cisco in 1996. The first version to see wide adoption was NetFlow v5 [120], which became available to the public around 2002. Although Cisco never published any official documentation on the protocol, the widespread use was in part result of Cisco making the corresponding data format freely available [2]. NetFlow v5 was obsoleted by the more flexible NetFlow v9, the state of which as of 2004 is described in [92]. NetFlow v9 introduced support for adaptable data formats through templates, as well as IPv6, Virtual Local Area Networks (VLANs) and MultiProtocol Label Switching (MPLS), among other features.

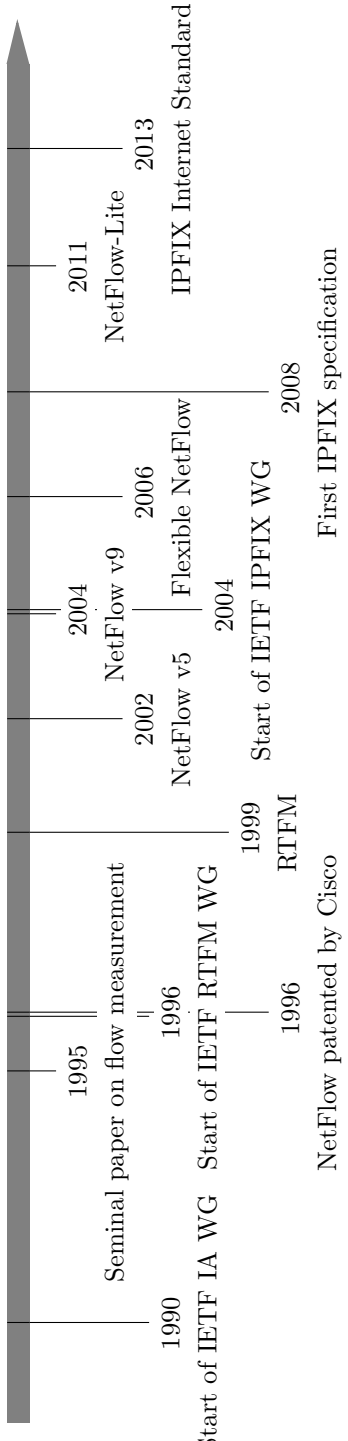


Figure 2.1: Evolution of flow export technologies and protocols.

Several vendors besides Cisco provide flow export technology alike NetFlow v9 (e.g., Juniper's J-Flow), which are mostly compatible with NetFlow v9. The flexibility in representation enabled by NetFlow v9 made other recent advances possible, such as more flexibility in terms of flow definitions. Cisco provides this functionality by means of its Flexible NetFlow technology [116]. Later, in 2011, Cisco presented NetFlow-Lite, a technology based on Flexible NetFlow that uses an external packet aggregation device to facilitate flow export on packet forwarding devices without flow export capabilities, such as datacenter switches [36].

Partly in parallel to the NetFlow development, the IETF decided in 2004 to standardize a flow export protocol, and chartered the IPFIX WG [129]. This WG first defined a set of requirements [91] and evaluated several candidate protocols. As part of this evaluation, NetFlow v9 was selected as the basis of the new IPFIX Protocol [93]. However, IPFIX is not merely "the standard version of NetFlow v9" [22], as it supports many new features. The first specifications were finalized in early 2008, four years after the IPFIX WG was chartered. These specifications were the basis of what has become the IPFIX Internet Standard [104] in late 2013. A short history on flow export and details on development and deployment of IPFIX are provided in [2].

Note that the term *NetFlow* itself is heavily overloaded in literature. It refers to multiple different versions of a Cisco-proprietary flow export protocol, of which there are also third-party compatible implementations. It refers as well to a flow export technology, consisting of a set of packet capture and flow metering implementations that use these export protocols. For this reason, we use the term *flow export* in this thesis to address exporting in general, without reference to a particular export protocol. As such, the term *NetFlow* is solely used for referencing the Cisco export protocol.

2.1.2 Related Technologies & Approaches

There are several related technologies with *flow* in the name that do not solve exactly the same problems as flow export. One is sFlow [142], an industry standard integrated into many packet forwarding devices for sampling packets and interface counters. Its capabilities for exporting packet data chunks and interface counters are not typical features of flow export technologies. Another difference is that flow export technologies also support 1:1 packet sampling, i.e., considering every packet for data export, which is not supported by sFlow. From an architectural perspective, which will be discussed in Section 2.2 for NetFlow and IPFIX, sFlow is however very similar to flow export technologies. Given its packet-oriented nature, sFlow is closer related to packet sampling techniques, such as the Packet SAMPling (PSAMP) standard [100] proposed by the IETF, than to a flow export technology. Given that this chapter is about flow export, we do not consider sFlow.

Another related technology, which is rapidly gaining attention in academia and network operations, is OpenFlow [134]. Being one particular technology

for Software-Defined Networking (SDN), it separates the control plane and data plane of networking devices [15]. OpenFlow should therefore be considered a flow-based configuration technology for packet forwarding devices, instead of a flow export technology. Although it was not specifically developed for the sake of data export and network monitoring, as is the case for flow export technologies, flow-level information available within the OpenFlow control plane (e.g., packet and byte counters) was recently used for performing network measurements [77]. Tutorials on OpenFlow are provided in [16], [66].

There exists also other flow export technology that is not related to and incompatible with NetFlow and IPFIX, but is designated to the same task of exporting network traffic flows. Argus [138] provides such technology, and is available as open-source software already since the early nineties. In contrast to software that is compatible with protocols like NetFlow and IPFIX, Argus uses a dedicated protocol for transferring flow data and therefore requires both Argus client and server software for deployment.

A data analysis approach that is often related to flow export is Deep Packet Inspection (DPI), which refers to the process of *analyzing* packet payloads. Two striking differences can be identified between DPI and flow export. First, flow export traditionally only considers packet headers, and is therefore considered less privacy-sensitive than DPI and packet export. Second, flow export is based on the aggregation of packets (into flows), while DPI and packet export are typically considering individual packets. Although seemingly opposing, we show throughout this chapter how DPI and flow export are increasingly united for increased visibility in networks.

2.2 Flow Monitoring Architecture

The architecture of typical flow monitoring setups consists of several stages, each of which is shown in Figure 2.2. The first stage is Packet Observation, in which packets are captured from an *Observation Point* and pre-processed. Observation Points can be line cards or interfaces of packet forwarding devices, for example. We discuss the Packet Observation stage in Section 2.3.

The second stage is Flow Metering & Export, which consists of both a *Metering Process* and an *Exporting Process*. Within the Metering Process, packets are aggregated into flows, which are defined as "sets of IP packets passing an observation point in the network during a certain time interval, such that all packets belonging to a particular flow have a set of common properties" [104]. After a flow is considered to have terminated, a flow record is exported by the Exporting Process, meaning that the record is placed in a datagram of the deployed flow export protocol. Flow records are defined in [104] as "information about a specific flow that was observed at an observation point", which may include both characteristic properties of a flow (e.g., IP addresses and port numbers) and measured properties (e.g., packet and byte counters). They can be imagined as records or

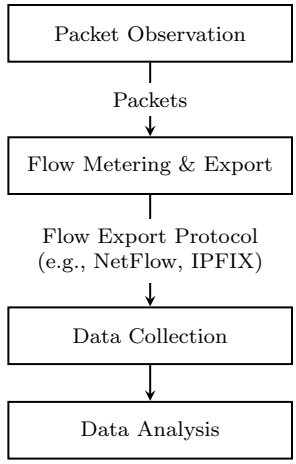


Figure 2.2: Architecture of a typical flow monitoring setup.

rows in a typical database, with one column per property. The *Metering* and *Exporting* processes are in practice closely related. We therefore discuss these processes together in Section 2.4.

The third stage is Data Collection, which is described in Section 2.5. Its main tasks are the reception, storage and pre-processing of flow data generated by the previous stage. Common pre-processing operations include aggregation, filtering, data compression, and summary generation.

The final stage is Data Analysis, of which two examples will be discussed in detail in Chapter 4 and 5. In research deployments, data analysis is often of an exploratory nature (i.e., manual analysis), while in operational environments, the analysis functions are often integrated into the Data Collection stage (i.e., both manual and automated). Common analysis functions include correlation and aggregation; traffic profiling, classification, and characterization; anomaly and intrusion detection; and search of archival data for forensic or other research purposes.

Note that the entities within the presented architecture are conceptual, and may be combined or separated in various ways, as we exemplify in Figure 2.3. We will highlight two important differences. First and most important, the Packet Observation and Flow Metering & Export stages are often combined in a single device, commonly referred to as *Flow Export Device* or *flow exporter*. When a flow exporter is a dedicated device, we refer to it as *flow probe*. Both situations are shown in Figure 2.3. We know however from our own experience that the IPFIX architecture [97] was developed with flow export from packet forwarding devices in mind. In this arrangement, packets are read directly from a monitored link or received via the forwarding mechanisms of a packet forwarding device. However, especially in research environments where trace data is ana-

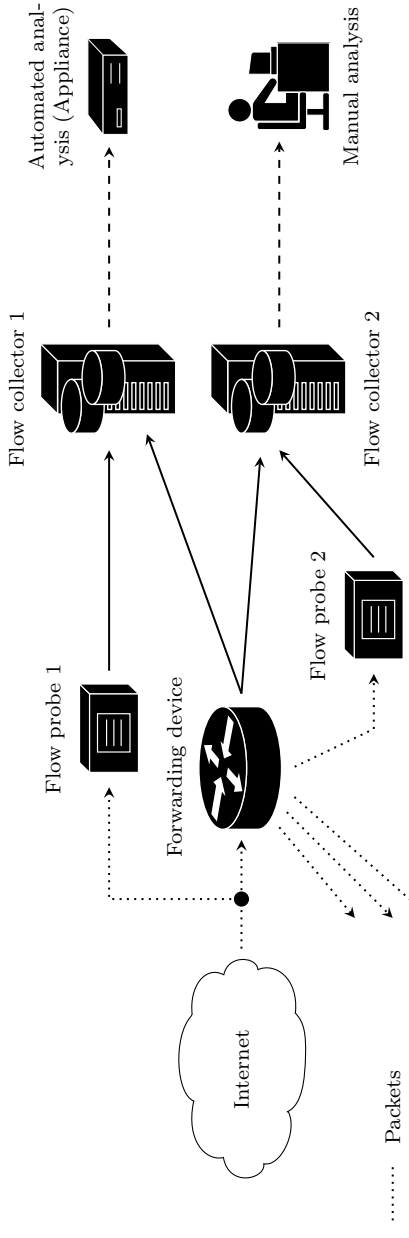


Figure 2.3: Various flow monitoring setups.

lyzed, packet capture may occur on a completely separate device, and as such should not be considered an integral part of the Flow Metering & Export stage. This is why we consider the Packet Observation and Flow Metering & Export stages in this work to be separate. A second difference with what is shown in Figure 2.2, is that multiple flow exporters can export flows to multiple devices for storage and pre-processing, commonly referred to as *flow collectors*. After pre-processing, flow data is available for analysis, which can be both automated (e.g., by means of an *appliance*) or manual.

2.3 Packet Observation

Packet observation is the process of capturing packets from the line and pre-processing them for further use. It is therefore key to flow monitoring. In this section, we cover all aspects of the Packet Observation stage, starting by presenting its architecture in Section 2.3.1. Understanding this architecture is however not enough for making sound packet captures; Also the installation and configuration of the capture equipment is crucial. This is explained in Section 2.3.2. Closely related to that are special packet capture technologies that help to increase the performance of capture equipment, which is surveyed in Section 2.3.3. Finally, in Section 2.3.4, we discuss one particular aspect of the Packet Observation stage in detail that is widely used in flow monitoring setups: packet sampling & filtering.

2.3.1 Architecture

A generic architecture of the Packet Observation stage is shown in Figure 2.4. Before any packet pre-processing can be performed, packets must be read from the line. This step, packet capture, is the first in the architecture and typically carried out by a Network Interface Card (NIC). Before packets are stored in on-card reception buffers and later moved to the receiving host's memory, they have to pass several checks when they enter the card, such as checksum error checks.

The second step is timestamping. Accurate packet timestamps are essential for many processing functions and analysis applications. For example, when packets from different observation points have to be merged into a single dataset, they will be ordered based on their timestamps. Timestamping performed in hardware upon packet arrival avoids delays as a consequence of forwarding latencies to software, resulting in an accuracy of up to 100 nanoseconds in the case of the IEEE 1588 protocol, or even better. Unfortunately, hardware-based timestamping is typically only available on special NICs using Field Programmable Gate Arrays (FPGAs), and most commodity cards perform timestamping in software. However, software-based clock synchronization by means of the Network Time Protocol (NTP) or the Simple Network Time Protocol (SNTP) usually provides

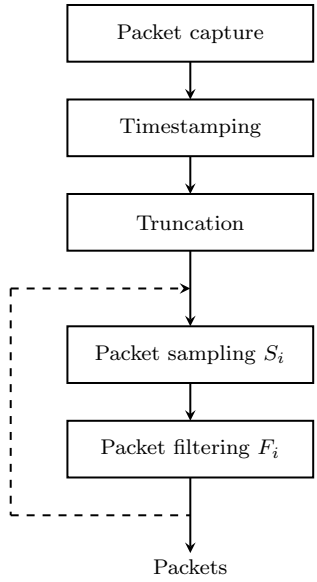


Figure 2.4: Architecture of the Packet Observation stage.

an accuracy in the order of 100 microseconds. For further reading, we recommend the overviews on time synchronization methods in [56], [59].

Both packet capture and timestamping are performed for all packets under any condition. All subsequent steps shown in Figure 2.4, are optional. The first of them is packet truncation, which selects only those bytes of a packet that fit into a preconfigured *snapshot length*. This reduces the amount of data received and processed by a capture application, and therefore also the number of computation cycles, bus bandwidth and memory used to process the network traffic. For example, flow exporters traditionally only rely on packet header fields and ignore packet payloads.

The last step of the Packet Observation stage is packet sampling and filtering [99]. Capture applications may define sampling and filtering rules so that only certain packets are selected for measurement. The motivation for sampling is to select a packet subset, while still being able to estimate properties of the full packet stream. The motivation for filtering is to remove all packets that are not of interest. Packet sampling & filtering will be discussed in detail in Section 2.3.4.

2.3.2 Installation & Configuration

In this subsection, we describe how packet captures should be made in wired, wireless, and virtual networks, and how the involved devices should be installed and configured. Most packet captures are made in wired networks, but can also

be made in wireless networks. Due to the popularity of virtual environments, packet captures in virtual networks are also becoming more common.

Most network traffic captures are made in wired networks, which can range from Local Area Networks (LANs) to backbone networks. This is mainly due to their high throughput and low external interference. Packet capture devices can be positioned in-line and in mirroring mode, which may have a significant impact on capture and network operation:

- *In-line mode* – The capture device is directly connected to the monitored link between two hosts. This can be achieved by installing additional hardware, such as bridging hosts or network taps [154]. Network taps[1] are designed to duplicate all traffic passing through the tap and provide a connection for a dedicated capture device. They use passive splitting (optical fiber networks) or regeneration (electrical copper networks) technology to pass through traffic at line rates without introducing delays or altering data. In addition, they have built-in fail open capability that ensures traffic continuity even if a tap stops working or loses power. Once a tap has been installed, capture devices can be connected or disconnected without affecting the monitored link. In Figure 2.3, *Flow probe 1* receives its input traffic by means of a network tap.

- *Mirroring mode* – Most packet forwarding devices can mirror packets from one or more ports to another port, to which a capture device is connected. This is commonly referred to as *port mirroring*, *port monitoring*, or *Switched Port ANalyzer (SPAN) session* [121]. Port mirroring requires a change in the forwarding device's configuration, but does not introduce additional costs as for a network tap. It should be noted that mirroring may introduce delays and jitter, alter the content of the traffic stream, or reorder packets [78]. In addition, care should be taken to select a mirror port with enough bandwidth; Given that most captures should cover two traffic directions (full-duplex), the mirror port should have twice the bandwidth of the monitored port, to avoid packet loss. In Figure 2.3, *Flow probe 2* receives its input traffic by means of port mirroring.

Packet captures in wireless networks can be made using any device equipped with a wireless NIC, under the condition that the wireless traffic is not encrypted at the link-layer, or the encryption key is known. Wireless NICs can however only capture at a single frequency at a given time. Although some cards support channel hopping, by means of which the card can switch rapidly through all radio channels, there is no guarantee that all packets are captured [86]. In large-scale wireless networks, it is more common to capture all traffic at a Wireless LAN (WLAN) Controller, which controls all individual wireless access points and forwards their traffic to other network segments by means of a high-speed wired

[1] Another commonly used name is Test Access Port (TAP).

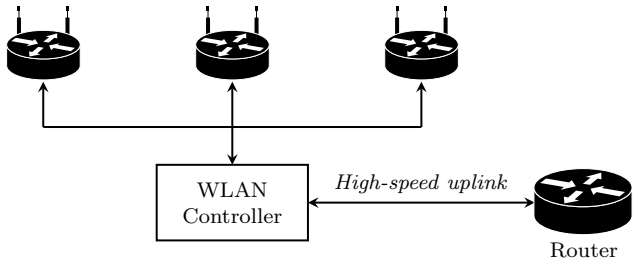


Figure 2.5: Packet capture in wireless networks.

interface. This is shown in Figure 2.5, where the high-speed uplink suitable with traffic from and to all access points can be captured. Besides having a single point of capture, the advantage is that link-layer encryption of wireless transmission protocols does not play a role anymore and captures can be made as described above for wired networks.

Deployment of packet capture devices in virtual networks is very similar to deployment in wired networks, and is rapidly gaining importance due to the widespread use of virtual machines (VMs). Virtual networks act as wired LANs, but are placed in virtual environments, e.g., to interconnect VMs. We therefore do not consider Virtual Private Networks (VPNs) as virtual networks, as they are typically just overlay networks in physical networks. Virtual networks use virtual switches [60], which support port mirroring and virtual taps. Furthermore, the mirrored traffic can be forwarded to dedicated physical ports and captured outside the virtual environment by a packet capture device.

Key to monitoring traffic is to identify meaningful observation points, ultimately allowing capture devices to gather most information on traffic passing by the observation point. These observation points should preferably be in wired networks. Even in wireless networks one should consider capturing at a WLAN controller to overcome all previously discussed limitations. In addition, deployment of network taps is usually preferred over the use of mirror ports, mainly due to effects on the packet trace of the latter. Port mirroring should only be used if necessary and is particularly useful for ad-hoc deployments and in networks where no taps have been installed.

2.3.3 Technologies

For most operating systems, libraries and Application Programming Interfaces (APIs) for capturing network traffic are available, such as *libpcap* or *libtrace* [1] for Linux and BSD-based operating systems, and *WinPcap* for Windows. These libraries provide both packet capture and filtering engines, and support reading from and writing to offline packet traces. From a technical point of view, they are located on top of the operating system's networking stack.

Since the networking stacks of operating systems are designed for general-purpose networking, packet captures usually suffer from suboptimal performance. The overall capture performance depends on system costs to hand over packets from the NIC to the capture application, via a packet capture library; Packets have to traverse several layers, which increase latency and limit the overall performance as they add per-packet processing overhead. Several methods have been proposed to speed up this process [30]:

- Interrupt mitigation and packet throttling (Linux NAPI) reduce performance degradation of the operating system under heavy network loads. Interrupt mitigation decreases the number of interrupts triggered by NICs during times of heavy traffic, as all interrupts convey the same message about a large number of packets waiting for processing. This reduces the system load. Packet throttling is applied when a system is overloaded with packets; Packets are already dropped by the NIC, even before they are moved to the software.
- Network stack bypass techniques, such as PF_RING, avoid the per-packet processing overhead caused by the various OS networking layers.
- Memory-map techniques for reducing the cost of copying packets form kernel-space to user-space, such as [29].

All these methods provide software-based optimizations for making packet captures. To be able to deal with higher packet rates, however, hardware-acceleration cards have been introduced. They use FPGAs to reduce CPU load during packet capture and guarantee packet capture without loss under modest CPU load [33]. Other capabilities of these cards are precise timestamping (with GPS synchronization), traffic filtering, and multi-core traffic distribution by means of multiple receive queues. They use Direct Memory Access (DMA) to receive and transmit packets. In that way, they also address the problem of passing packets efficiently from NICs to the capture application.

Modern commodity NICs provide a cost-effective solution for making high performance packet captures on links with speeds up to 10 Gbps [24]. Features provided by controllers on these NICs (e.g., Intel 82599, Myri-10G Lanai Z8ES) include multiple receive queues by means of *Receive Side Scaling* to distribute packets across multiple CPUs [40], and a DMA engine to off-load CPU processing. To be able to use these features, vendors provide a set of libraries and drivers for fast packet processing, such as Intel DPDK[2] and PF_RING DNA/Libzero[3].

It is important to fully understand the performance of the packet capture process to create and operate reliable monitoring applications. Care should be taken when selecting a packet capture library or system: Most packet capture benchmarks show throughputs for situations without any further processing, which

[2] `http://www.dpdk.org/`

[3] `http://www.ntop.org/products/pf_ring/libzero-for-dna/`

may overestimate the real performance when some form of packet processing is used. An example of such packet processing is flow export, which will be discussed in the subsequent sections. Key to high-performance packet processing is efficient memory management, low-level hardware interaction, and application optimizations.

2.3.4 Packet Sampling & Filtering

The objective of packet sampling and filtering is to forward only certain packets to the Flow Metering & Export stage. A combination of several sampling and filtering steps can be adopted to select the packets of interest.

Packet sampling[4] aims at reducing the load of subsequent stages or processes (e.g., the *Metering Process* within the Flow Metering & Export stage) and, consequently, to reduce the consumption of bandwidth, memory and computation cycles. Therefore, sampling should be used whenever it is expected that the number of monitored packets will overload the following stage. The ultimate objective is to turn the uncontrolled loss of packets caused by overload into a controlled one by using sampling.

Several sampling strategies are defined in [99], where two major classes of sampling schemes can be distinguished: *Systematic sampling* schemes deterministically decide which packets are selected (for example every Nth packet in a periodic sampling scheme). In contrast, *random sampling* selects packets in accordance to a random process. The main challenge when using sampling is to obtain a representative subset of the relevant packets. In general, random sampling should be preferred over systematic sampling when both are available, because the latter can introduce unwanted correlations in the observed data. For example, a measurement using periodic sampling would be likely biased towards or against periodic traffic. This restriction can be relaxed when it is known in advance that the monitored traffic is highly aggregated, i.e., it comprises of traffic from many different hosts, applications, etc. In such a situation, the influence of the sampling scheme is less noticeable, although its quantitative impact on the resulting flow data depends on the nature of the traffic [9].

Packet sampling obviously entails loss of information. Depending on the employed sampling scheme, some properties of the original packet stream can be easily recovered. For example, if a simple random sampling scheme is used, the total number of packets or bytes can be estimated by multiplying the measured numbers by the inverse of the sampling probability. Reciprocally, it means that sampling with a rate of 1:N results in a reduction of load to the *Metering Process* (in terms of number of packets and bytes to process) by a factor of N. Other characteristics of the original data are affected in a more complex way. For example, longer flows are more likely to be sampled than shorter ones. A simple scaling would yield a biased estimation of the flow length distribution.

[4] See [27] and [8] for an introduction to sampling in the context of network management.

Methods to estimate sampled flow statistics have been discussed in [9]. Several publications propose new sampling schemes that aim at mitigating the effects of sampling, for example by automatically adapting the sampling rate according to the traffic load [39].

The role of *packet filtering* is to deterministically "separate all the packets having a certain property from those not having it" [99]. Similar to sampling, filtering can be used to reduce the amount of data to be processed by the subsequent stages. Again, two major classes can be distinguished: With *Property Match Filtering*, a packet is selected if specific fields within the packet (and/or the router state) are equal to a specified value or inside a specified value range. Typically, such filters are used to limit packet capturing to specific IP addresses, applications, etc. *Hash-Based Filtering* applies a hash function to the packet content or some portion of it, and compares the result to a specified value or value range. Hash-Based Filtering can be used to efficiently select packets with common properties or, if the hash function is applied to a large portion of the packet content, to select packets quasi-randomly.

2.4 Flow Metering & Export

The Flow Metering & Export stage is where packets are aggregated into flows and flow records are exported, which makes it key to any flow monitoring system. Its architecture is shown in Figure 2.6. The packet aggregation is performed within the *Metering Process*, based on Information Elements that define the layout of a flow. Information Elements are discussed in Section 2.4.1. After aggregation, an entry per flow is stored in a *flow cache*, explained in Section 2.4.2, until a flow is considered to have terminated and the entry is expired. Expiration of flow cache entries is discussed in Section 2.4.3. After one or more optional flow-based sampling and filtering functions, which are discussed in Section 2.4.4, flow records have to be encapsulated in messages. This is where IPFIX comes in, which is defined in [22] as "a unidirectional, transport-independent protocol with flexible data representation". IPFIX message structures and types are discussed in Section 2.4.5. Finally, a transport protocol has to be selected, which is discussed in Section 2.4.6. For an extensive analysis of open-source and commercial flow metering and export implementations, we refer the reader to [11].

2.4.1 Information Elements

Fields that can be exported in IPFIX flow records are named IEs. The Internet Assigned Numbers Authority (IANA) maintains a standard list of IEs as the IPFIX Information Element registry [127]. Use of the IANA registry for common IEs is key to cross-vendor operability in IPFIX. Besides IANA IEs, enterprise-specific IEs can be defined, allowing for new fields to be specified for a particular application without any alterations to IANA's registry. IEs have a name, nu-

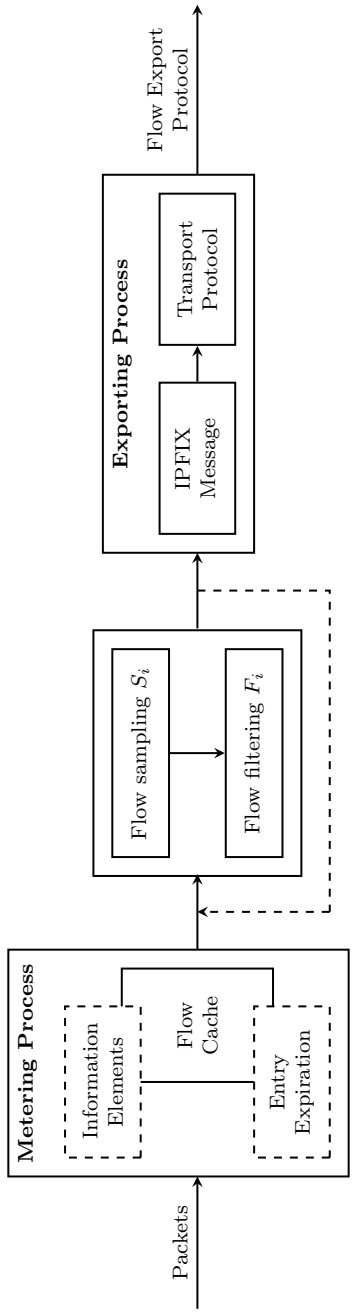


Figure 2.6: Architecture of the Flow Metering & Export stage.

ID	Name	Description
152	flowStartMilliseconds	Timestamp of the flow's first packet.
153	flowEndMilliseconds	Timestamp of the flow's last packet.
8	sourceIPv4Address	IPv4 source address in the packet header.
12	destinationIPv4Address	IPv4 destination address in the packet header.
7	sourceTransportPort	Source port in the transport header.
11	destinationTransportPort	Destination port in the transport header.
4	protocolIdentifier	IP protocol number in the packet header.
2	packetDeltaCount	Number of packets for the flow.
1	octetDeltaCount	Number of octets for the flow.

Table 2.1: Examples of common IPFIX Information Elements, from [127].

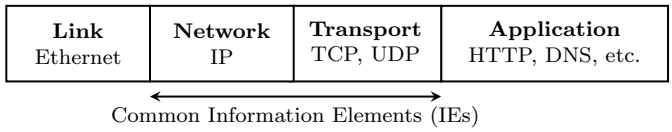


Figure 2.7: Network layers considered for IEs.

meric ID, description, type, length (fixed or variable), and status (i.e., *current* and *deprecated*), together with an enterprise ID in the case of enterprise-specific IEs [105].

A subset of IEs defined in [127] is shown in Table 2.1, which is often considered the smallest set of IEs for describing a flow. These IEs are for transport-layer and network-layer fields, and supported by most flow exporters. However, in contrast to what the name "IP flow information eXport" (IPFIX) suggests, IEs can be defined for any layer, ranging from the link-layer (L2) up to and including the application-layer (L7), as shown in Figure 2.7. For example, IEs have been defined for Ethernet [46], such as `sourceMacAddress` (ID 56) and `vlanID` (ID 58). We refer to the support for application-layer IEs as *application awareness*. In other words, flow exporters with application awareness combine DPI with traditional flow export.

There are also other IEs that are different from the default transport- and network-layer IEs shown in Table 2.1 in terms of type and semantic. For example, since many IEs are identical to what can be retrieved using widely used Simple Network Management Protocol (SNMP) Management Information Base (MIB) variables, such as `interfaceName` (ID 82), a current standardization effort is working to define a method for exporting SNMP MIB variables using IPFIX [109]. This avoids the repetitive definition of IEs in the IANA registry. Another example are IEs for exporting octet sequences, such as `ipPayloadPacketSection` (ID 314), which can be useful for exporting sampled packet chunks.

Guidelines on the definition of globally unique IEs are provided in [106], which are intended for both those defining new IEs and reviewing the specifications of new IEs. Before defining a new IE, one should be sure to define an IE that 1) is unique within the IANA IE registry, 2) is self-contained, and 3) represents nonproprietary information. After definition, the IE specification should be sent to IANA, after which the request for approval is evaluated by a group of experts, named "IE-Doctors". Upon approval, IANA is requested to apply the necessary changes to the IE registry. The same process applies to requests for IE deprecation.

The configuration of Metering Processes in terms of IEs is not standardized and varies from exporter to exporter. However, flow collectors are always instructed by flow exporters by means of *templates*, which are used to describe which IEs are used for which flow. This approach is also used by NetFlow v9, although it is not compatible with IPFIX, because of the different message formats used by the two protocols. NetFlow v5 does not provide template support and is therefore fixed to its initial specification. This considerably limits the applicability of NetFlow v5, since no protocol evolution is possible. NetFlow v5 cannot be used for monitoring IPv6 traffic, for example. It is however often suggested that NetFlow v5 is the most widely deployed flow export protocol and therefore still a relevant source of flow information [7], [136].

In addition to what has been described before, several more advanced mechanisms with respect to IEs have been defined: variable-length encoding and structured data. Variable-length encoding can be used for IEs with a variable length in IPFIX, despite of IPFIX' template mechanism being optimized for fixed-length IEs [104]. As such, longer IEs can be transferred efficiently since no bytes are wasted due to a fixed-size reservation. Structured data in IPFIX [102] is useful for transferring a list of the same IE, by encapsulating it in a single field. A clear use-case for this are MPLS labels; Since every packet can carry a stacked set of such labels, one traditionally has to define a dedicated IE for every label position, e.g., `mplsTopLabelStackSection`, `mplsTopLabelStackSection2`, etc. With structured data, an MPLS label stack can be encoded using a single IE.

2.4.2 Flow Caches

Flow caches are tables within a Metering Process that store information regarding all active network traffic flows. These caches have entries that are composed of IEs, each of which can be either a key or non-key field. The set of key fields, commonly referred to as the *flow key*, is used to determine whether a packet belongs to a flow already in the cache or to a new flow, and therefore defines which packets are grouped into which flow. In other words, the flow key defines the properties that distinguish flows. Incoming packets are hashed based on the flow key and matched against existing flow cache entries. A match triggers a flow cache update where packet and byte counters are tallied. Packets not matching any existing entry in the flow cache are used to create new entries. Commonly

used key fields are source and destination IP addresses and port numbers. Non-key fields are used for collecting flow characteristics, such as packet and byte counters.

Given that source and destination IP addresses are normally part of the flow key, flows are usually unidirectional. In situations where both forward and reverse flows (between a source/destination pair) are important, bidirectional flows [95] may be considered. Bidirectional flow records have counters for both directions, and a special IE (`biflowDirection`, ID 239) to indicate a flow's initiator and responder. Since source and destination IP addresses are still part of the flow key in a setup for bidirectional flows, special flow cache support is needed for identifying matching forward and reverse flows.

Several parameters should be considered when selecting or configuring a flow cache for a particular deployment, such as the cache layout, type and size. The flow cache layout should match the selection of key and non-key fields, as these are the IEs accounted for each flow. Given that there are many types of IEs available, flow cache layouts should be able to cope with this flexibility. For example, application information in flow records is becoming more and more important, which can be concluded both from the fact that IEs are being registered for applications in IANA's IE registry, as well as flow exporters with application identification support are being developed. Flow caches, thus, should support flexible flow definitions per application.

Flow caches can also differ from each other in terms of type. For example, IPFIX defines flows that consist of a single packet, commonly referred to as *single-packet flows*[5] [97]. A regular flow cache typically cannot be used for single-packet flows, as the cache management (e.g., the process that determines which flow has terminated) of such caches is often not fast enough. To overcome this problem, some vendors implement dedicated caches for such flows, sometimes referred to as *immediate cache* [117]. An example use case for single-packet flows and immediate caches is a configuration with a very low packet sampling rate, such as 1:2048, where it is expected that no more than one packet is sampled per flow. In those situations, one can avoid resource-intensive cache management by using an *immediate cache*. Besides caches for single-packet flows, it is possible to use caches from which flow entries cannot expire, but are periodically exported, named *permanent cache* [117]. These caches can be used for simple flow accounting, as they do not require a flow collector for collecting flow records; As flow cache entries are never expired, packet and byte counters are never reset upon expiration and therefore represent the flow state since the Metering Process has started.

The size of flow caches depends on the memory available in a flow exporter and should be configured/selected based on the expected number of flows, the selected key and non-key fields, and expiration policies. Given that expiration

[5] In terms of expiration, which is discussed in Section 2.4.3, these flows are said to have a zero timeout.

policies have the strongest impact on the required flow cache size, we discuss them in the next subsection.

2.4.3 Flow Cache Entry Expiration

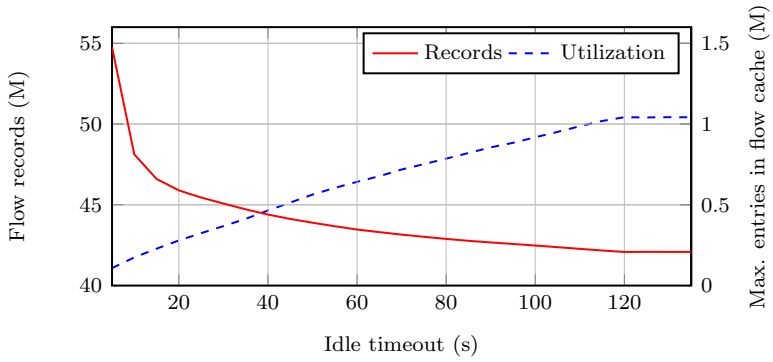
Cache entries are maintained in the flow cache until the corresponding flows are considered to have terminated, after which the entries are expired. These entries are usually expired by the Metering Process according to given timeout parameters or when specific events have occurred. IPFIX, however, does not mandate precisely when flow entries need to be expired and flow records exported. Instead, it provides the following reasons as guidelines on how Metering Processes should expire flow cache entries [97]:

- *Active timeout* – The flow has been active for a specified period of time. Therefore, the active timeout helps to report the activity of long-lived flows periodically. Typical timeout values range from 120 seconds to 30 minutes. Note that cache entries expired using the active timeout are not removed from the cache; Counters are reset, and start and end times are updated.
- *Idle timeout* – No packets belonging to a flow have been observed for a specified period of time. Typical timeout values range from 15 seconds to 5 minutes.
- *Resource constraints* – Special heuristics, such as the automatic reduction of timeout parameters at run-time, can be used to expire flows prematurely in case of resource constraints.

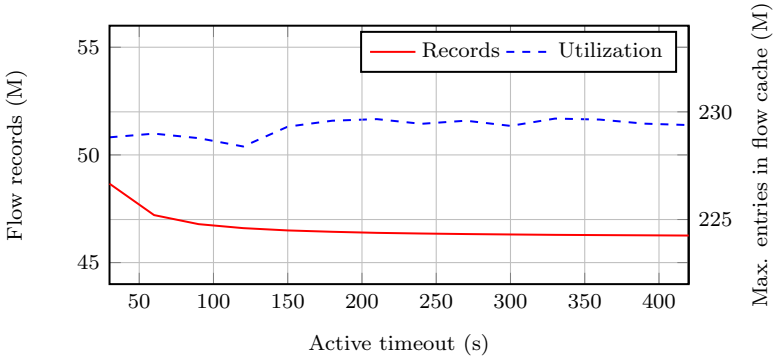
Other reasons for expiring flow cache entries can be found in various flow exporter implementations:

- *Natural expiration* – A TCP packet with a `FIN` or `RST` flag set has been observed for a flow and therefore the flow is considered to have terminated.
- *Emergency expiration* – A number of flow entries are immediately expired when the flow cache becomes full.
- *Cache flush* – All flow cache entries have to be expired in unexpected situations, such as a significant change in flow exporter system time after time synchronization.

The configured active and idle timeout values have impact on the total number of flow records exported for a particular dataset and the number of flow entries in the flow cache. On the one hand, using longer timeout values results in a higher aggregation of packets into flow records, which is generally positive and desirable to reduce the load on flow collectors. On the other hand, using longer timeout values means that it takes longer before a flow becomes visible to the Data Analysis stage.



(a) Varying idle timeout values, active timeout = 120 seconds



(b) Varying active timeout values, idle timeout = 15 seconds

Figure 2.8: Impact of timeouts on the aggregation of packets into flows and flow cache utilization.

To illustrate the expiration behavior of a typical flow exporter, we have performed several experiments on the campus network of the UT. These experiments are based on a packet trace consisting of one day of network traffic between the campus network of the UT and the Dutch NREN SURFnet, accounting for 2.1 TB of data. Our experiments have been performed on the impact of active and idle timeout values on 1) the number of resulting flow records, and 2) the maximum flow cache utilization. *nProbe* has been used for exporting the flows without sampling. All experiments have been performed twice: Once by varying the active timeout value while maintaining a fixed idle timeout value, and once by varying the idle timeout value while maintaining a fixed active timeout value.

The experiment results are shown in Figure 2.8. The figure shows the maximum number of concurrently used flow cache entries for various timeout values. Several conclusions can be derived from the experiment results. First, as shown in

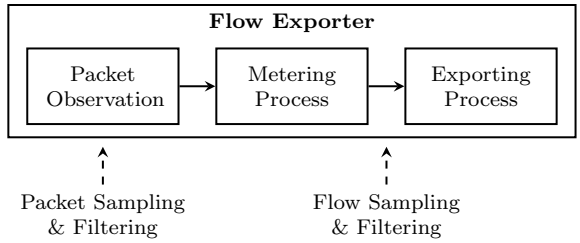


Figure 2.9: Sampling & filtering in a flow exporter.

Figure 2.8a, an increasing idle timeout value results in fewer flow records, which is the case because of more packets being aggregated into the same flow record. This implies that flow entries stay in the flow cache for a longer time, resulting in a higher flow cache utilization. Second, the number of exported flow records and the maximum flow cache utilization stabilize for an increasing active timeout value, as shown in Figure 2.8b. This can be explained by the fact that most flow entries are expired by the idle timeout because of the very large active timeout value. Third, as soon as the idle timeout value equals the active timeout value (i.e., 120 seconds for our experiments), as shown in Figure 2.8a, the number of flow records and the flow cache utilization stabilize again, which is due to the fact that flow records are expired by the active timeout. We have also measured the impact of using natural expiration based on TCP flags and conclude that it barely affects the total number of flow records and the flow cache utilization.

Besides showing the relation between active and idle timeout behavior, the results in Figure 2.8 provide insight into the minimum flow cache size required for monitoring the link in this specific example, which has a top throughput of roughly 2 Gbps. For example, given an active and idle timeout values of 120 and 15 seconds, respectively, the maximum flow cache utilization never exceeds 230k cache entries.

2.4.4 Flow Record Sampling & Filtering

Flow record sampling and filtering provide a means to select a subset of flow records, to reduce the processing requirements of the *Exporting Process* and all subsequent stages. In contrast to packet sampling and filtering, which are performed as part of the Packet Observation stage, flow record sampling and filtering functions are performed after the Metering Process and therefore work on flow records instead of packets. This is shown in Figure 2.9. As a consequence, either all packets of a flow are accounted, or none.

The techniques for performing flow record sampling and filtering are similar to packet sampling and filtering, which have been described in Section 2.3.4. We distinguish again between systematic sampling and random sampling [107]. Systematic sampling decides deterministically whether a flow record is selected (for

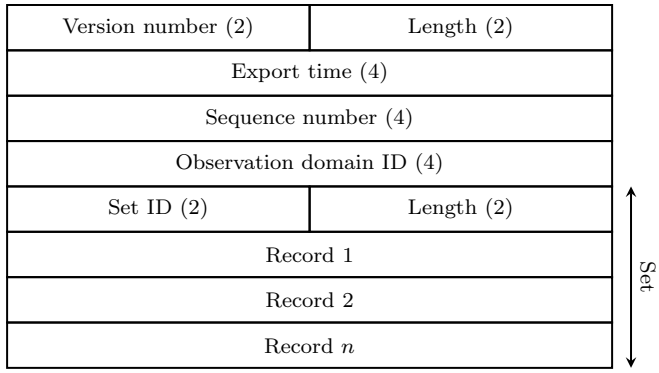


Figure 2.10: IPFIX message (simplified) [104].

example, every Nth flow record in periodic sampling). In contrast, with random sampling, flow records are selected in accordance to a random process. As for packet sampling, random sampling should be generally preferred over systematic sampling when in doubt about the characteristics of the traffic, because the latter can introduce unwanted correlations in the observed data.

Flow record filtering can be distinguished between *Property Match Filtering* and *Hash-Based Filtering* [107]. Property Match Filtering for flow records works similarly to Property Match Filtering for packets, but rather than filtering on packet attributes, filtering is performed on flow record attributes. It is particularly useful when only flow records for specific hosts, subnets, ports, etc. are of interest. With Hash-Based Filtering, flow records are selected if the hash value of their flow key lies within a predefined range of values. Hash-Based Filtering can be used for selecting a group of flow records from different observation points. Flow records from different observation points can be correlated because the flow key shared by packets belonging to the same flow results in the same hash value.

2.4.5 IPFIX Messages

A simplified version of the IPFIX message format [104] is shown in Figure 2.10. The field size in bytes is shown for fields with a fixed size; Other fields have a variable length. The first 16 bytes of the message form the message header and include a protocol version number, message length, export time and an *observation domain ID*. After the header come one or more *Sets*, which have an ID and a variable length, and can be of any of the following types:

- *Template Sets* contain one or more templates, used to describe the layout of Data Records.
- *Data Sets* are used for carrying exported Data Records (i.e., flow records).

- *Options Template Sets* are used for sending meta-data to flow collectors, such as control plane data or data applicable to multiple Data Records [96]. For example, Options Template Sets can be used to inform flow collectors about the flow keys used by the Metering Process.

Sets are composed of one or more records. The number of records in an IPFIX message is usually limited to avoid IP fragmentation. It is up to the Exporting Process to decide how many Records make up a message, while ensuring that the message size never exceeds the Maximum Transmission Unit (MTU) of a link (e.g., 1500 bytes) [104]. An exception to this rule is a situation in which IEs with variable lengths that exceed the link Maximum Transmission Unit (MTU) are exported.

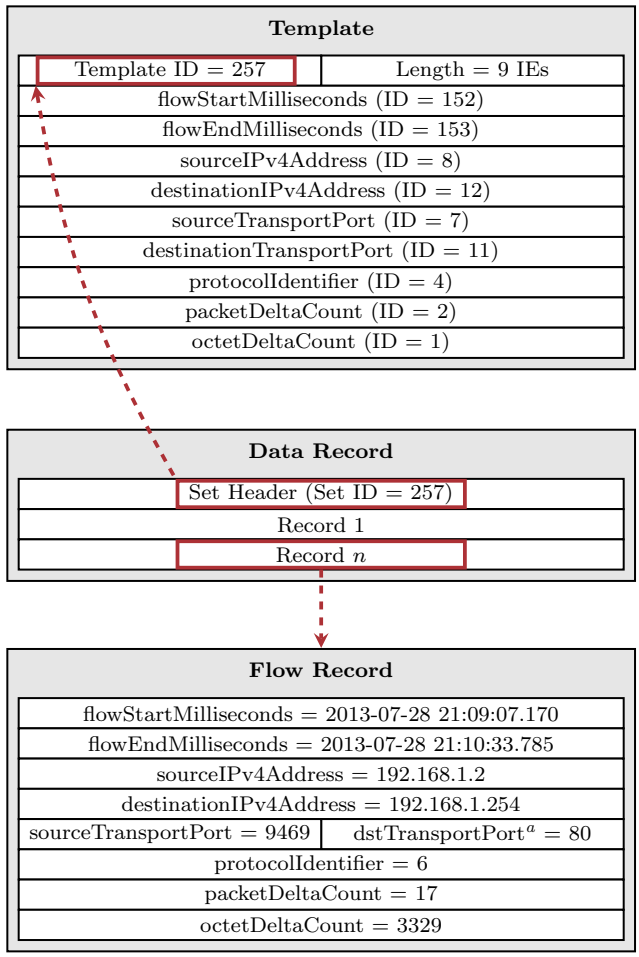


Figure 2.11: Correlation between IPFIX data types (simplified) [104].

[a]This IE has been abbreviated for the sake of space. The full IE name is shown in the template.

	SCTP	TCP	UDP
Congestion awareness	+	+	–
Deployability	–	+	+
Graceful degradation	+	–	–
Reliability	+	+	–
Secure transport	+	+	–

Table 2.2: Comparison of transport protocols for IPFIX.

An example of a template, a corresponding Data Record, and a flow record is shown in Figure 2.11. The template is shown at the top of the figure, and consists of an ID (257) and 9 IEs. A corresponding Data Record points at the appropriate template by listing its ID. This is mandatory to provide a means for flow collectors to associate Data Records with their templates. Also multiple flow records are included in the Data Record, which must adhere to the full set of IEs listed in the template.

2.4.6 Transport Protocols

After constructing an IPFIX message for transmission to a flow collector, a transport protocol has to be selected. A key feature of IPFIX is support for multiple transport protocols [104]. A comparison of transport protocols for IPFIX is provided in Table 2.2, where '+' stands for *supported* or *good*, and '-' for *unsupported* or *poor*.

The Stream Control Transmission Protocol (SCTP) [94] is the mandatory transport protocol to implement for IPFIX. It provides a congestion-aware and sequential packet delivery service; Packets are kept in sequence as with TCP, and packet boundaries are preserved as with UDP, i.e., the receiver can distinguish between individual packets, rather than a stream of bytes as with TCP. SCTP also provides multiple streams per connection, which can be used to avoid head-of-line blocking when multiple logical separate streams (e.g., one per template) are exported simultaneously [103]. The partial reliability extension [90] to SCTP provides further flexibility: The Exporting Process can cancel retransmission of unreceived datagrams after a given timeout. This allows graceful degradation via selective dropping of exported datagrams under high load, rather than overloading buffers with pending retransmissions.

Despite these advantages, SCTP is currently the least-deployed of the three supported protocols. The reason is primarily a practical one: IPFIX over SCTP can be difficult to implement, mainly because support for SCTP lacks on other operating systems than Linux and BSD. Bindings to DTLS[6] for secure transport may also be hard to find in all but the most recent versions of TLS libraries.

[6]DTLS is an implementation of TLS for transmission over datagram transport protocols, such as UDP and SCTP.

There are also deployment considerations. Since much more effort has gone into TCP stack optimization than SCTP stack optimization, the latter can be slower than TCP. It can also be difficult to send SCTP packets across the Internet, as some middleboxes drop SCTP packets as having an unrecognized IP protocol number. Also, many Network Address Translation (NAT) devices often fail to support SCTP. However, given its advantages, we advocate using SCTP for flow export in every situation in which it is possible to do so.

IPFIX supports transport over TCP as well. TCP provides congestion-aware, reliable stream transport. It is widely implemented and, as such, it is very easy to implement IPFIX over TCP on most platforms. Bindings to TLS for secure transport are also widely available, which makes IPFIX over TLS over TCP the preferred transport for exporting flow records over the Internet. The primary problem with IPFIX over TCP is that TCP does not degrade gracefully in overload situations. Specifically, the TCP receiver window mechanism limits the Exporting Process' sending rate when the Collecting Process is not ready to receive, thereby locking the rate of export to the rate of collection. This pushes buffering back to the Exporting Process, which is generally the least able to buffer datagrams. Careful selection of TCP socket receive buffer sizes and careful implementation of the Collecting Process can mitigate this problem, but those implementing Collecting Processes should be aware of it.

The most widely implemented and deployed transport protocol for flow export is UDP. UDP has the advantage of being easy to implement even in hardware Exporting Processes. It incurs almost no overhead, but on its turn provides almost no service: "Best-effort" (or "unreliable") delivery of packets without congestion awareness. As a consequence, UDP should be used for flow export with care. The lack of any congestion awareness means that high-volume export may incur significant loss. The lack of flow control means that Collecting Processes must use very large socket buffers to handle bursts of flow records. As the volume of exported flow records increases dramatically during Denial of Service (DoS) attacks or other incidents involving large numbers of very short flows, the lack of flow control also may make UDP futile for measuring such incidents. Another serious problem concerns templates. On UDP, Exporting Processes must periodically resend templates to ensure that Collecting Processes have received them. While IPFIX does provide a sequence numbering facility to allow a Collecting Process to roughly estimate how many flow records have been lost during export over UDP, this does not protect templates. A Collecting Process that loses a template, or that restarts in the middle of an export, may be unable to interpret any flow records until the next template retransmission.

A fourth method provided by IPFIX is the IPFIX File Format [101]. IPFIX messages are grouped together into files, which can be stored and transported using any of the various protocols that deal with files (e.g., SSH, HTTP, FTP and NFS). File transport is not particularly interoperable and therefore not recommended in general. However, it may be worth considering in specific cases, such as making IPFIX flow data widely available via a well-known URL.

2.5 Data Collection

Flow collectors are an integral part of flow monitoring setups, as they receive, store, and pre-process[7] flow data from one or more flow exporters in the network. Data collection is performed by one or more Collecting Processes within flow collectors. Common pre-processing tasks are data compression [41], [101], aggregation [108], data anonymization, filtering, and summary generation.

In this section, we discuss the most important characteristics of flow collectors. We start by describing the various formats in which flow data can be stored in Section 2.5.1. After that, in Section 2.5.2, we provide best-practices in the field of data anonymization. Anonymization is a type of data obfuscation that ensures anonymity of individuals and prevents tracking individual activity. For an extensive analysis of open-source and commercial flow collection implementations, we refer the reader to [11].

2.5.1 Storage Formats

The functionality and performance provided by flow collectors depend strongly on the underlying data storage format, as this defines how and at which speed data can be read and written. This section discusses and compares the various available storage formats, which should allow one to choose a flow collector that satisfies the requirements of a particular setup or application area. We can distinguish two types of storage formats:

- *Volatile* – Volatile storage is performed in-memory and therefore very fast. It can be useful for data processing or caching, before it is written to persistent storage.
- *Persistent* – Persistent storage is used for storing data for a longer time and usually has a larger capacity. However, it is significantly slower than volatile storage.

Although flow data often has to be stored for a long time (e.g., to comply with data retention laws), it can be useful to keep data in-memory. This is mostly the case when flow data has to be analyzed on-the-fly, and only results have to be stored. In those situations, one can benefit from the high performance of volatile storage. An example use case is the generation of a time-series in which only the time-series data itself has to be stored.

When data has to be stored beyond the time needed for processing, it has to be moved to persistent storage. This, however, results in a bottleneck, due to the difference in speed between volatile and persistent storage. Depending on the system facilitating the flow collection, one may consider to compress data before

[7] We talk about *pre-processing* here, as we assume that *processing* is done in the Data Analysis stage.

	Flat files	Row-oriented databases	Column-oriented databases
Disk space	+	–	0
Insertion performance	+	–	0
Portability	– (binary), + (text)	–	–
Query flexibility	–	+	+
Query performance	+ (binary), – (text)	–	+

Table 2.3: Comparison of data storage formats.

moving it to persistent storage (more details are provided in Section 2.6.3). We distinguish between the following types of persistent storage:

- *Flat files* – Flat file storage is usually very fast when reading and writing files, while providing limited query facilities [35]. Examples of flat file storage are binary and text files.

- *Row-oriented databases* – Row-oriented databases store data in tables by means of rows and are frequently used in Database Management Systems (DBMSs), such as MySQL[8], PostgreSQL[9], and Microsoft SQL Server[10]. Accessing the data is therefore done by reading the full rows, even though only part of the data may be needed to answer a query.

- *Column-oriented databases* – Column-oriented databases, such as FastBit[11], store data by column rather than by row. Only fields that are necessary for answering a query are therefore accessed.

A comparison of these data storage formats is shown in Table 2.3, where '+' stands for *good*, '–' for *poor*, and '0' for *average*. We evaluate each format based on disk space requirements, insertion performance, portability, query flexibility and query performance. We consider *nfdump* a representative option for binary flat file storage, MySQL for row-oriented databases, and FastBit for column-oriented databases.

In terms of disk space, flat files have a clear advantage above the database-based approaches. This is mainly due to the fact that row- and column-oriented databases usually need indexes for shorter query response times, which consume more disk space on top of the "raw" dataset. For example, it is described in [45]

[8] `https://www.mysql.com/`
[9] `http://www.postgresql.org/`
[10] `http://www.microsoft.com/sql/`
[11] `https://sdm.lbl.gov/fastbit/`

that MySQL with indexes needs almost twice the capacity of *nfdump* for a particular dataset. For the case of FastBit, it is shown to be less capacity-intensive than MySQL (depending on its configuration) [35], but more than *nfdump*. High compression rates can be achieved since data in a particular column is usually very similar, i.e., homogeneous. The highest insertion performance can be achieved using flat files, as new data can be simply added to the end of a file, without any additional management overhead, such as updating indexes in the case of MySQL. When it comes to portability, text-based flat files have the clear advantage of being readable by many tools on any system. However, flat files usually provide only limited query language vocabulary, which makes databases more flexible in terms of possible queries.

In contrast to database-based approaches, flat file storage is usually not indexed; Sequential scans over datasets are therefore unavoidable. However, since many flat file storages create smaller files at regular intervals, this can be considered a coarse time-based index that limits the size of sequential scans by selecting fewer input files in a query. Several works have compared the performance of the various data storage formats in the context of flow data collection. The performance of binary flat files and MySQL is compared in [45], where query response times are measured for a set of queries on subsets of a single dataset. The authors show that binary storage outperforms MySQL-based storage in all tested scenarios and advocate the use of binary storage when short query response times are required. A similar methodology has been used in [73], where the performance of FastBit is compared with binary flat files. It is shown that FastBit easily outperforms binary storage in terms of query response times, which is explained by the fact that FastBit only reads columns that are needed for a query. This results in fewer Input/Output (I/O) operations. The performance of FastBit- and MySQL has been compared in [35], where the authors conclude that FastBit-based storage is at least an order of magnitude faster than MySQL.

The performance of the described approaches can generally be improved by distributing flow data over multiple devices. For example, it is a common practice to use storage based on a Redundant Array of Independent Disks (RAID) in a flow collector, which ensures data distribution over multiple hard drives in a transparent fashion. Even more performance improvements can be achieved by deploying multiple flow collectors and distributing the data between them. This, however, requires some management system that decides how data is distributed (for an example, see [57]).

2.5.2 Data Anonymization

Flow data traditionally has a significant privacy protection advantage over raw or sampled packet traces: Since flow data generally does not contain any payload, the content of end-user communications is protected. However, flows can still be used to identify individuals and track individual activity and, as such, the collection and analysis of flow data can pose severe risks for the privacy of end

users. The legal and regulatory aspects of this privacy risk, and requirements to mitigate it are out of scope for this work – these are largely a matter of national law, and can vary widely from jurisdiction to jurisdiction. Instead of surveying the landscape of data protection laws, we make a general simplifying assumption that IP addresses can be used to identify individuals and as such should be protected. Other information available in flows can be used to disambiguate flows and therefore may be used to profile end users or to break IP address anonymization.

Best practices for trace data anonymization are laid out by CAIDA in [113], drawing on the state of the art in anonymization techniques surveyed in [137]. The key tradeoff in IP address anonymization is between privacy risk and data utility. There is no 'one-size-fits-all' flow data anonymization strategy, as data utility is also dependent on the type of analysis being done. For example, simply removing IP address information carries with it the lowest risk of identification, but also makes the data useless for anything requiring linkage of flows to hosts; For simple statistics on flow durations and volumes, for example, such data can however still be useful.

In the more general case, since networks are structured, a structure-preserving anonymization technique such as the Crypto-PAn algorithm [10] allows anonymized IP addresses to be grouped by prefix into anonymized networks. This preserves significant utility for analysis, at the cost of restricting the space of possible anonymized addresses for a given real address, making profiling attacks against address anonymization easier. Even given this tradeoff, the significantly increased utility of the results leads to a recommendation for Crypto-PAn.

Given the restricted space of solutions to the anonymization problem, it has become apparent that unrestricted publication of anonymized datasets is probably not a tenable approach to the sharing of flow data [3], as attacks against anonymization techniques scale more easily than strengthening these techniques while maintaining utility. Technical approaches to data protection are therefore only one part of the puzzle; Non-technical protection, such as sharing of data in vetted communities of researchers, or analysis architectures whereby analysis code is sent to a data repository and only results are returned, must also play a part in preserving the privacy of network end-users.

2.6 Lessons Learned

In the previous sections, we have discussed how to setup a typical flow monitoring system. Before the exported flow data can however be used for production or measurement purposes, the setup has to be verified and calibrated. This section discusses how hidden problems that impact the resulting data and are caused by infrastructural problems, can be detected and potentially overcome. We start by discussing flow exporter overload in Section 2.6.1, followed by transport overload and flow collector overload in Section 2.6.2 and 2.6.3, respectively. Another source

of problems with flow data are measurement artifacts, typically related to flow exporter implementations, which will be covered in detail in Chapter 3.

2.6.1 Flow Exporter Overload

Flow caches in a flow exporter usually have a fixed size that is either constrained by hardware, or determined at compile-time in the case of flow exporter software. When this size turns out to be small, flow data loss or low performance can be the result. The latter is especially the case for software-based solutions, as they often use linked lists to store different flow cache entries under the same hash value, which results in longer cache entry lookup times. Given that it is not always possible to foresee significant changes in the monitored traffic, flow caches may eventually turn out under-dimensioned.

Since under-dimensioned flow caches usually result in data loss it is important to be aware of this happening, especially when the exported flow data is used for critical applications. For those having access to a flow exporter, be it a packet forwarding device or dedicated probe, it is usually trivial to obtain these data loss statistics. For example, software exporters often write them to log files, while the flow cache utilization and loss statistics of hardware flow exporters can be obtained via a Command-Line Interface (CLI), or SNMP. An example of how to retrieve such details from Cisco switches is provided in Section 3.2. For those having only access to the exported flow data, it is much harder to derive conclusions about data loss. An example of this is also provided in Section 3.4, where it is shown that indications of an under-dimensioned flow cache can be retrieved from the dataset with some uncertainty.

Several actions can be taken to reduce flow exporter load, without the need to replace a flow exporter. First, expiration timeouts can be reduced. Especially the idle timeout should be considered here, as it expires cache entries of flows that are inactive anyway. Although reducing timeouts results in a lower flow cache utilization, which has been shown in Section 2.4.3, this will result in more flow records. Care should be taken to not overload a flow exporter's Exporting Process or a flow collector with this larger number of flow records.

A second action for reducing the load of flow exporters is enabling packet sampling or decreasing the packet sampling rate, i.e., reducing the number of packets forwarded to the Metering Process. We have measured the impact of packet sampling on the resulting flow data based on the dataset presented in the introductory words of this section. The results are shown in Figure 2.12 and two conclusions can be derived from this figure. First, the use of packet sampling does not necessarily reduce the number of flow records, mostly because of the use of timeout-based expiration and the nature of the traffic; When intermediate packets in a flow are not sampled and therefore not considered by the Metering Process, this flow can be split into multiple flows by the applied idle timeout. This demonstrates that packet sampling reduces the load of the Metering Process, but not necessarily of the subsequent stages. Second, the impact of packet

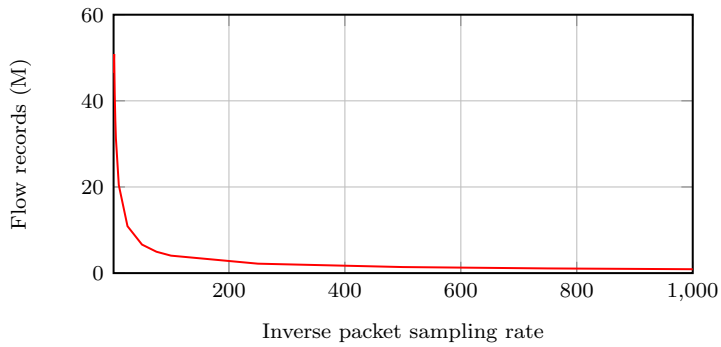


Figure 2.12: Impact of packet sampling on the number of flow records.

sampling on the number of flow records reduces when the sampling rate is increased (except for very high rates, such as 1:2, in our dataset). In contrast to packet sampling, flow sampling reduces the number of flow records, but it will not help to reduce utilization of the flow cache, as it is applied after the Metering Process. Enabling packet sampling or increasing its rate however results in information loss that is in various cases very hard to (mathematically) compensate for. A flow-based anomaly detection system, for example, which is based on thresholds determined on a non-sampled dataset or datasets captured using another sampling rate, will function sub-optimally or stop functioning completely. It is therefore advised to tune flow entry expiration first, before enabling or modifying sampling parameters.

As soon as a flow exporter is experiencing capacity problems due to resource constraints, flow records may start to be expired in a different way. That means, the active and idle timeouts are respected as much as possible, until the utilization of the flow cache exceeds a threshold (e.g., 90%). At that moment, an emergency expiration kicks in which expires cache entries prematurely, to free up the flow cache and to allow new entries to be inserted. This results in more flow records, and flow data that is not expired consistently, which may impact the subsequent Data Analysis stage. For example, an IDS that counts the number of flow records for detecting a particular anomaly may not function properly anymore by raising false alerts. It is therefore important to be aware of these dynamics in flow export setups.

2.6.2 Transport Overload

It is not uncommon for flow exporters to export data from links of 10 Gbps and higher over links of 1 Gbps. This is usually the case because flow data is sent to collectors of the exporter's management interface (which usually has a line speed of 1 Gbps), and because flow collectors are often "normal" file servers that are not

Sampling rate	Protocol	Export packets / bytes
1:1	NetFlow v5	1.4 M / 2.1 G
1:1	NetFlow v9	3.5 M / 2.5 G
1:10		1.6 M / 1.1 G
1:100		314.9 k / 222.5 M
1:1000		72.2 k / 49.5 M
1:1	IPFIX	4.3 M / 3.0 G

Table 2.4: Export volumes for the UT dataset (2.1 TB).

equipped with special, high-speed network interfaces. Due to the data reduction achieved by flow export, flow data exported from high-speed links can generally be exported over smaller links without data loss. However, in anomalous situations as described in the previous subsection, such links will become a bottleneck in a flow monitoring system. This is especially the case in anomalous situations, where the data aggregation achieved by flow export is constrained, such as under DDoS attacks. One type of DDoS attack are flooding attacks, which aim at overloading targets by opening many new connections by sending a large number of TCP `SYN`-packets, for example. Due to the definition of a flow, this type of traffic results in many new flows, because of the changing flow key in every packet [64]. Moreover, depending on the selected number of IEs for each flow, the data aggregation usually achieved can change into data export that is neutral in size (i.e. the exported data has the same size as the monitored data) or even data amplification; As soon as the overhead of the IPFIX Message is larger than the monitored packets on the line (e.g., during a flooding attack), amplification takes place. This is not uncommon, as many flow exporters use by default a set of IEs that is larger in terms of bytes than a TCP `SYN`-packet, for example.

We have measured the volume of NetFlow and IPFIX Messages in terms of packet and bytes for the UT dataset introduced in Section 2.4.3. The results for various packet sampling rates and export protocols are shown in Table 2.4. Two main observations can be made. First, the newer the flow export protocol, the more packets and bytes are sent to the flow collector. This can be explained by the inclusion of (option) templates in NetFlow v9 and IPFIX, and the increase in timestamp size from 32 bits relative timestamps (in NetFlow v9) to 64 bits absolute timestamps[12] in IPFIX. Second, regardless of the export protocol used, NetFlow and IPFIX traffic is roughly 0.1% of the original traffic in a setup without packet sampling. This is in contrast to [96], where it is claimed that the IPFIX traffic generated by a flow exporter is 2-5% of the traffic of the monitored link.

Both NetFlow and IPFIX carry sequence numbers in their respective packet headers, which assist in identifying packet loss when either unreliable trans-

[12]IPFIX supports timestamps at multiple granularities, ranging from seconds to nanoseconds. The used flow exporter, *nProbe*, uses millisecond resolution timestamps by default for IPFIX.

ports are used (e.g., UDP) or the transport bandwidth is permanently underdimensioned. Flow collectors, such as *nfdump*, keep track of these sequence numbers and store the number of sequence failures in the metadata of each file. Before deriving any conclusion from sequence failure counters, attention should be paid to the export protocol version. Both NetFlow v5 and IPFIX provide sequence numbers in terms of flow records, while NetFlow v9 provides these in terms of export packets. As such, the actual number of missing flow records can only be estimated when NetFlow v5 or IPFIX are used.

2.6.3 Flow Collector Overload

Flow data collection usually gets least attention of all stages in a typical flow monitoring setup, although a suboptimal setup can result in severe – and often unnoticed – data loss. What makes it difficult to dimension a flow collector, is that in anomalous situations, the number of incoming flow records can be doubled or even more than that. Considerable over-provisioning and calibration are therefore needed to ensure that no data is lost. In this subsection, we discuss how flow collectors should be calibrated by means of performance measurements.

Data loss as part of flow collector overload can be a consequence of both kernel and application buffer overflows, and disk I/O overload. Kernel buffers store NetFlow and IPFIX Messages before they are received by a Collecting Process. Application buffers are part of the flow collector itself and can be used for any intermediate processing before the flow data is stored. Buffers can be tuned to a certain extent; Increasing buffer sizes allows more data to be temporarily stored, but is useless if subsequent system elements are becoming a bottleneck. In practice, disk I/O will be a bottleneck in such situations, so increasing buffers provides only limited advantages.

Many flow collectors apply data compression to flow data by default. Whether or not compression should be enabled depends on the processing and storage capacities of the system acting as a flow collector. As a rule of thumb, one can compare the time needed to store a flow data chunk both in compressed and uncompressed format. If writing compressed data is faster, the storage subsystem is the bottleneck of the collection system and data compression, which is CPU-intensive, should be enabled. Otherwise, if writing uncompressed data is faster, processing capacity is the bottleneck and data compression should be disabled. This rule of thumb is confirmed by *nftest*, part of *nfdump*, which performs these tests automatically and provides one with an advice on whether or not to enable data compression.

To get an idea of how much processing capacity is needed to store flow data of one day, we have performed several measurements on our UT dataset, where we have used *nfdump* as the flow collector software. The storage volumes for various export parameters are listed in Table 2.5. The listed storage volumes are for compressed datasets and are slightly less than 1 GB per day when no packet sampling is used. To put this in contrast; The Czech NREN CESNET does not

Sampling rate	Protocol	Storage volume	Reduction factor
1:1	NetFlow v5	912.7 MB	2301x
1:1	NetFlow v9	1.0 GB	2100x
1:10		503.7 MB	4169x
1:100		103.9 MB	20212x
1:1000		20.4 MB	102941x
1:1	IPFIX	820.4 MB	2560x

Table 2.5: Storage volumes for the UT dataset (2.1 TB).

apply packet sampling either and stores roughly 125 GB of flow data per day, while SURFnet, the Dutch NREN, stores around 16 GB per day with a packet sampling rate of 1:100.

2.7 Conclusions

This chapter has shown and discussed all aspects of a full-fledged flow monitoring setup based on NetFlow or IPFIX, covering the complete spectrum of packet observation, flow metering and export, data collection, and data analysis. The generic architecture of such setups, as well as the various stages it involves, have been explained in Section 2.2. Additionally, we have shown that each of these stages affects the final flow data and consequently, its analysis. Understanding all these stages is therefore of key importance to anyone performing flow measurements, and for anyone using flow data for compromise detection in particular.

Based on the results presented in this chapter, we conclude that flow measurements can be used for compromise detection if two requirements are satisfied. First, the exported flow data should reflect the original network traffic flows precisely. Although this appears trivial for any flow data analysis, there are various factors that we have explored in this chapter that cause the exported data to not be a precise representation of the original network traffic. For example, packet sampling results in packets not being metered at all, resulting in an unfaithful representation of the original flows by definition. Also, the incorrect configuration of expiration timers may result in multiple flows being merged into the same flow record, especially if port numbers are reused (e.g., during large attacks resulting in many connections). Taking the default active timeout value of Cisco of 30 minutes, for example, would yield many flows to be merged into the same record. Second, we have shown that overload may occur in various stages of flow monitoring systems. During attacks that involve many small connections, the meta-data used for flow accounting may even result in amplification of the original data. We investigate this in detail in Chapter 6. Given that overload typically results in data loss, care must be taken to dimension the flow monitoring system appropriately.

CHAPTER 3

Flow Measurement Artifacts

Flow data provides an aggregated view of network traffic in which streams of packets are grouped in flows. The resulting scalability gain usually excuses the coarser data granularity, as long as the flow data reflects the actual network traffic precisely. For many applications, such as accounting and profiling, flow data is a proven and reliable source of information. However, we have observed that the flow export process may introduce artifacts in the exported data, which may impair more advanced data analyses, such as compromise detection. In this chapter, we therefore investigate several artifacts we discovered to impair flow data analysis, by explaining how they can be detected in datasets and which implementation decisions are causing them. In addition, we verify the artifacts' presence in data from a set of widely-used devices. Our results show that the revealed artifacts are widely spread among different devices from various vendors. Based on our observations, we conclude whether compromise detection based on flow data is feasible in practice, complementary to our theoretical analysis in Chapter 2.

The paper related to this chapter is [49], which received a Best Paper Award at PAM 2013.

The organization of this chapter is as follows:

- *Section 3.1 introduces existing works in the context of flow data artifacts.*
- *Section 3.2 describes the case study that forms the basis of the artifact analysis presented in this chapter, by means of the Cisco Catalyst 6500. This device is widely deployed in many service provider, enterprise and campus networks, and selected as the basis of our analysis.*
- *Section 3.3 explains our experiment setup and describes the set of devices from various vendors that was involved in our analysis.*
- *Section 3.4 discusses results of our artifact analysis for all devices involved in our analysis.*
- *Section 3.5 concludes this chapter.*

3.1 Related Work

Analyses have shown that the advantages offered by flow export often come at the expense of accuracy, although the gains of using flow data normally excuse this. Since IPFIX is still in its early days and NetFlow deployment is far more mature [70], most literature on flow data artifacts is about NetFlow (especially NetFlow v9).

Flow data artifacts described in literature can be classified into three categories:

- Timing, related to the way in which flow exporters put timestamps into flow records, how these timestamps are stored by export protocols, and how precise flow exporters are in expiring flow records.
- Data loss, causing irreparable damage to flow datasets.
- Minor inaccuracies that can usually be repaired or ignored.

Artifacts in the first category, timing, are all related to the way in which NetFlow accounts flow record start and end times. NetFlow v9 in particular uses two separate clocks: an uptime clock in milliseconds that is used to express flow record start and end times in terms of a flow exporter's uptime, and a real-time clock (UNIX time) that is used to map those uptimes to an absolute time. The real time is inserted in NetFlow packets together with the uptime before they are transmitted to a flow collector. It is then up to a flow collector to calculate the absolute start and end times of flow records based on these two types of timestamps. The advantage of using only a single real-time clock is that there are no two clocks that need to remain synchronized all the times. However, several artifacts related to timing have been reported in literature. First, it is explained in [53] and [72] that the millisecond-level precision of the flow exporter uptimes is sacrificed, since only second-level timestamps of the real time can be stored in a NetFlow v9 packet. This leads to imprecise or incorrect start and end times of flow records. The same works describe that both clocks are not necessarily synchronized, resulting in a clock skew in the order of seconds per day. In addition, two timestamps are not necessarily inserted into the NetFlow packet at exactly the same moment either due to resource exhaustion or explicit export rate limiting, resulting in an additional delay [72]. Another category of timing artifacts is the imprecise or erroneous expiration of flow records, resulting in periodic patterns in the flow dataset and erroneously merged flow records [32].

The second category of artifacts is related to data loss. For example, it is described in [32] that gaps may be present in flow data, usually caused by an under-dimensioned flow cache, as described in Section 2.6.1. As a consequence, packets cannot always be accounted to flow records in the flow cache, effectively resulting in data loss.

The remainder of this chapter describes five more artifacts that extend the set of known artifacts in the second and third category. Also, where possible, artifacts reported in related works are verified.

3.2 Case Study: Cisco Catalyst 6500

The Cisco Catalyst 6500 is a widely deployed series of switches that can be found in many service provider, enterprise and campus networks [123]. In this section, we discuss five artifacts that are present in flow data from a specific device of this series[1], located in the campus network of the UT. This knowledge is therefore gained from our operational experience. It is important to note that this list is by no means comprehensive, since artifacts are load- and configuration-dependent. Moreover, artifacts related to clock imprecisions discussed by previous works, which we have observed as well, are not covered.

Imprecise Flow Record Expiration – Expiration is the process of removing flow records from the *NetFlow table* (i.e., flow cache). This can be done for a variety of reasons, such as timeouts and exporter overload. However, according to the documentation, flow records can be expired as much as 4 seconds earlier or later than the configured timeout [115] when the device is not overloaded. Moreover, the *average* expiration deviation should be within 2 seconds of the configured value. This is because of the way in which the expiration process is implemented: A software process scans the *NetFlow table* for expired flow records. Due to the time needed for scanning all flow records, expiration is often pre- or postponed.

TCP Flows Without Flag Information – TCP flags are accounted for few TCP flows, since they are solely exported for software-switched flows [115]. These flows are processed by a generic Central Processing Unit (CPU), while hardware-switched flows are processed using Application Specific Integrated Circuitss (ASICs). Whether a flow has been switched in hardware or software can be concluded from the *engineID* field in the flow records. Since most packets are hardware-switched, only few TCP flows with flags can be found in the exported data. Another observation can be made regarding the handling of flags of hardware-switched TCP flows: In contrast to what is specified in [115], TCP FIN and RST flags trigger the expiration of flow records. As such, TCP flags are considered in the expiration process, even though they are not exported.

Invalid Byte Counters – It has been observed before that byte counters in flow records are not always correct [53]. The counters represent the number of bytes associated with an IP flow [92], which is the sum of IP packet header and payload sizes. IP packets are usually transported as Ethernet payload, which should have a minimum size of 46 bytes according to IEEE 802.3-2008. If the

[1]The exact configuration can be found in Table 3.1 (Exporter 1).

payload of an Ethernet frame is less than 46 bytes, *padding bytes* must be added to fill up the frame. However, stripping these *padding bytes* is not done for hardware-switched flows, resulting in too many reported bytes.

Non-TCP Flow Records With TCP ACK Flag Set – The first packet of a new flow is subject to Access Control List (ACL) checks, while subsequent packets bypass them for the sake of speed. Bypassing ACL checks could also be done by fragmenting packets, since packet fragments are not evaluated. To overcome this security problem, Cisco has implemented a poorly documented solution that has two implications on software-switched flows. Firstly, flag information in flow records is set to zero for all packet fragments, which are always software-switched. Secondly, flag information in flow records of all other software-switched traffic is set to a non-zero value, and TCP ACK was chosen for that purpose.

Gaps – Similarly to the devices analyzed in [32], this exporter often measures no flows during short time intervals. This is caused mostly by hardware limitations, combined with a configuration that is not well adjusted to the load of the network. When a packet has to be matched to a flow record, its key fields are hashed and a lookup is done in a lookup table (*NetFlow TCAM*). In our setup, both the lookup table and the table storing the flow records (*NetFlow table*) consist of 128k entries with a hash efficiency of 90%, resulting in a net utilization of roughly 115k entries. A table (named *alias CAM* or *ICAM*) with only 128 entries is available to handle hash collisions, so that up to two flows with different keys but identical hashes can be stored. The event in which a packet belonging to a new flow cannot be accommodated because of hash collisions is called *flow learn failure*. The evolution of *flow learn failures* in this device can be monitored using the `CISCO-SWITCH-ENGINE-MIB` (SNMP).

3.3 Experiment Setup

To understand whether the artifacts presented in the previous section can also be identified in flow data from other flow exporters, several devices from three vendors, installed in campus and backbone networks throughout Europe, have been analyzed. Table 3.1 lists these devices, together with their hardware configuration and software (versions). Given the variety of hardware configurations, we cover a wide range of hardware revisions of widely used devices.

The first two devices, both from the Cisco Catalyst 6500 series, have identical hardware configurations and similar software versions, but are exposed to different traffic loads. We can therefore analyze whether the load of these devices affects the presence of artifacts. The third Cisco Catalyst 6500 has a significantly different hardware configuration and software version. The Cisco Catalyst 7600 series, represented by our fourth device, is generally similar to the Cisco Catalyst 6500 series, but uses different hardware modules. Device 1, 2 and 4 use the same hardware implementation of NetFlow (EARL7), while Device 3 is signifi-

Exporter	Model	Modules	Software
1	Cisco Catalyst 6500	WS-SUP720-3B (PFC3B, MSFC3)	IOS 12.2(33)SXI5
2	Cisco Catalyst 6500	WS-SUP720-3B (PFC3B, MSFC3)	IOS 12.2(33)SXI2a
3	Cisco Catalyst 6500	VS-SUP2T-10G-XL (PFC4XL, MSFC5) + WS-X6904-40G	IOS 15.0(1)SY1
4	Cisco Catalyst 7600	RSP720-3C-GE (PFC3C, MSFC4)	IOS 15.2(1)S
5	Juniper T1600	MultiServices PIC 500	JUNOS 10.4R8.5
6	INVEA-TECH FlowMon Probe	–	FlowMon Probe 3.01.02

Table 3.1: Assessed flow exporters and their configurations.

cantly newer (released in 2012) and uses Cisco's EARL8 ASIC. The fifth analyzed device is a Juniper T1600, which has also been analyzed in [32]. The inclusion of this device allows us to extend the results in [32]. Finally, we have included a dedicated flow exporter (probe) from INVEA-TECH. In the remainder of this chapter, we denote each of the devices by its number in the table.

3.4 Artifact Analysis

Section 3.2 described a set of artifacts present in flow data from a Cisco Catalyst 6500 (Exporter 1). This section evaluates whether these artifacts are also present in flow data from the other exporters listed in Section 3.3. For each artifact, we define the experiment methodology, followed by a description of our observations in both flow and SNMP data. After that, we show some examples in which the artifacts have impact on specific analysis applications. Also, we discuss whether the artifacts are repairable or not.

Imprecise Flow Record Expiration – Flow exporters are expected to expire flow records at the configured active timeout T_{active} and idle timeout T_{idle}, and possibly after a packet with TCP FIN or RST flag set has been observed. We perform the following experiments to evaluate the behavior of the flow exporters:

- **Active timeout:** We send a series of packets with identical flow key to the flow exporter for a period of $T_{active} + d$. The inter-arrival time of the packets is chosen to be less than T_{idle}. The experiment is performed for $d = -2, -1, \ldots, 16$ seconds. For each value of d, we repeat the experiment 100 times and count how often the flow exporter generates two flow records from the received packets. Ideally, one should see only one flow record per experiment for $d < 0$ and always two flow records per experiment for $d \geq 0$.

- **Idle timeout:** We send two packets with identical flow key to the exporter, separated by a time difference of $T_{idle} + d$. The rest of the experiment is performed as for the active timeout. Again, one ideally sees only one flow record per experiment for $d < 0$ and always two flow records for $d \geq 0$.

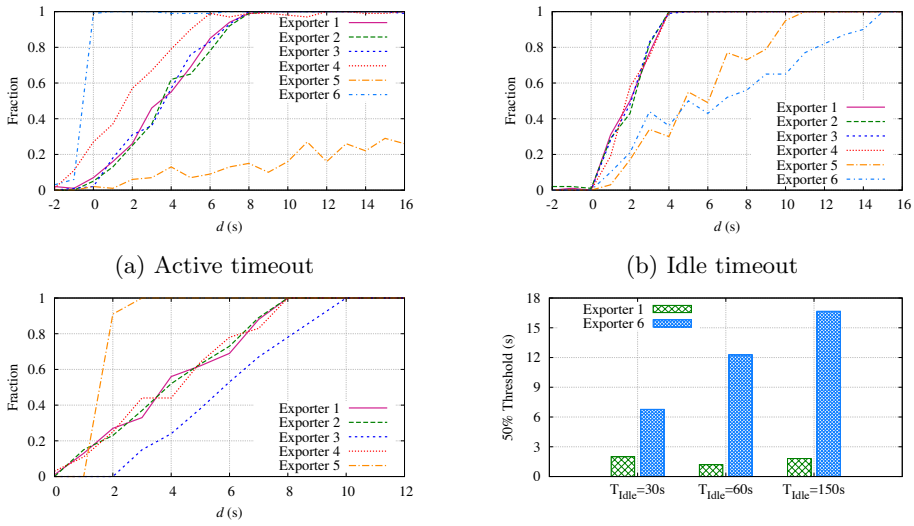
- **TCP FIN/RST flag:** We send one packet with the FIN or RST flag set, followed by another packet after d time units. The rest of the experiment is performed as for the active timeout (only for $d = 0, 1, \ldots, 16$). Ideally, the exporter always generates two flow records.

For all experiments, the packets are generated such that they are processed in hardware by the exporter, if applicable[2]. In addition, several initial packets are generated where necessary, to avoid that special mechanisms for the early expiration of records of small and short flows (such as Cisco's *fast aging* [115]) are applied. All exporters use an active timeout between 120 and 128 seconds, and an idle timeout between 30 and 32 seconds. Note that we do not rely on the timestamps in flow records, which means that we are not susceptible to the errors described in [72]. Instead, we use the system time of the hosts running the measurement scripts, which are placed close to the analyzed exporters.

The experiment results are shown in Figure 3.1a-3.1c for the three expiration mechanisms, respectively. For each value of d (in seconds, on the X-axis) we give the fraction of experiment runs (on the Y-axis) for which the flow exporter has generated two flow records. With regard to the active timeout (Figure 3.1a), Exporter 1-3 behave similarly: The number of experiments with two flow records increases linearly for $d \in [0, 8]$. Although this timespan of 8 seconds is in line with Cisco's documentation, the center of the timespan is incorrect: Instead of being at $d = 0$, our experiments show that it is at $d = 4$. Exporter 4 behaves similarly to the previous exporters, although the linear increase takes place for $d \in [-2, 6]$. Exporter 5 shows unexpected behavior: Even for $d = 16$, only 20% of the experiments result in two flow records. Additional experiments have shown that the expiration does not stabilize at all. Moreover, incorrect start times are reported for flow records expired by the active timeout (which confirms the findings in [32]). Finally, only Exporter 6 works as expected and always generates two flow records for $d \geq 0$.

The results obtained from the idle timeout experiments are shown in Figure 3.1b. Exporter 1-4 show identical behavior and the linear increase of the curve for $d \in [0, 4]$ confirms that the flow record expiration works according to its specification [115]. Exporter 5 performs better compared to the active timeout experiments: For $d \geq 11$ always two flow records are generated, which is in line with the findings in [32]. Flow records from Exporter 6 are expired up to 15s after the idle timeout, approximately linearly with $d \in [0, 15]$. We have observed that the behavior of this exporter also depends on the absolute value of the idle

[2] `http://www.cisco.com/en/US/products/hw/switches/ps708/products_tech_note09186a00804916e0.shtml`



(a) Active timeout

(b) Idle timeout

(c) TCP flags

(d) Impact of idle timeout

Figure 3.1: Results of the flow record expiration experiments.

timeout. In Figure 3.1d, we show for different idle timeouts the value of d (on the Y-axis) where 50% of the experiments yield two flow records, comparing the behavior of Exporter 1 and Exporter 6. While these values are always close to $2s$ for Exporter 1, they increase with the timeout for Exporter 6.

Figure 3.1c shows the results for the expiration based on TCP flags. The expiration behavior of Exporter 3 differs from the other Cisco devices, due to a different implementation of NetFlow (see Section 3.3). Overall, the number of correctly exported flow records increases linearly with d. The deviation d for which Exporter 5 wrongly exports only one flow record, is small: Three seconds after the FIN/RST flag was sent, always two records are exported. Exporter 6 does not expire flow records based on TCP flags by specification.

The flow record expiration behavior of Exporter 1-4 shows a clear linear slope in Figure 3.1a-3.1c, which suggests the presence of a cyclic process to expire and export the (hardware) flow tables. The fact that flow records are not expired exactly on the defined timeouts may not be a problem if flows are aggregated afterward. This is especially the case for flow records expired by the active timeout. However, when the idle timeout or TCP flags are used to signal the end of a flow, this artifact may result in irreparable data damage. For example, in [67] it is shown that some applications (e.g., peer-to-peer clients) often reuse sockets shortly after a TCP connection attempt failure. When timeouts and TCP flags are not observed strictly, packets from different connections may be merged into a single flow record.

Exporter	TCP Flows Without Flag Information	Invalid Byte Counters
1, 2	No flags exported for hardware-switched flows	Invalid byte counters for hardware-switched flows
3	Flags exported	
4	No flags exported for hardware-switched flows	
5, 6	Flags exported	Byte counters OK

Table 3.2: Artifact analysis results.

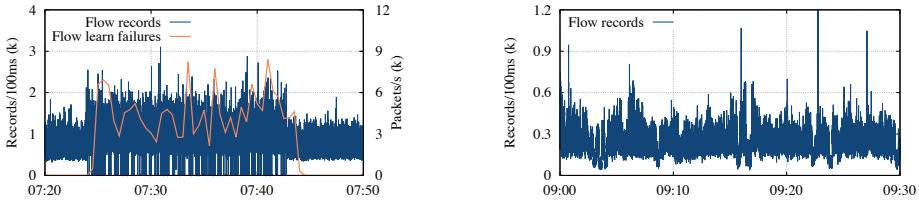
TCP Flows Without Flag Information – Our analysis results for this artifact are summarized in Table 3.2. The oldest assessed devices, Exporter 1, 2 and 4, do not export flags for hardware-switched TCP flows. Since the vast majority of flows is hardware-switched, TCP flags are rarely exported. We have observed that approximately 99.6% of all TCP flow records exported by Exporter 1 and 2 have no flag information set during a measurement period of one week. However, flags are respected for flow record expiration, even in the case of hardware-switched TCP flows. In the case of Exporter 3, 5 and 6, TCP flags are exported.

The lack of TCP flag information in flow records can be problematic for several types of data analysis. From a network operation perspective, TCP connection summaries can help to identify connectivity or health problems of services and devices. From a research perspective, many works rely on TCP connection state information. For example, [9], [38], [43] use it for inferring statistics from sampled flow data and [54] for optimizing sampling strategies. None of these approaches works on flow data without TCP flags.

Invalid Byte Counters – The results for this artifact are also summarized in Table 3.2. None of the Cisco devices strips the padding bytes from Ethernet frames of hardware-switched flows. Exporter 5 and 6 strip these bytes properly. The impact of this artifact depends on the fraction of Ethernet frames that carry less than 46 bytes of payload. To understand the distribution of packet sizes in current networks, we analyzed a packet trace of the UT campus network (1 day in 2011), and a trace from the CAIDA 'equinix-sanjose' backbone link[3] (1 day in 2012). In both traces, around 20% of the frames contains less than 46 bytes of payload, which would be reported incorrectly. The number of incorrectly counted bytes lies around 0.2% of the total number of bytes in both cases. The impact of this artifact on accounting applications is, therefore, very small.

Non-TCP Flow Records With TCP `ACK` Flag Set – Our analysis has shown that only flow data from older Cisco devices (*i.e.*, Exporter 1, 2 and 4) contains this artifact. On average, the number of non-TCP flow records with TCP `ACK` flag set accounts for approximately 1% of the total number of flow records on Exporter 1 and 2.

[3]The CAIDA UCSD Anonymized Internet Traces 2012 - 16 February 2012
`http://www.caida.org/data/passive/passive_2012_dataset.xml`



(a) *Flow learn failures* (b) NetFlow packet loss

Figure 3.2: Impact of *flow learn failures* and NetFlow packet loss on flow time-series.

When analysis applications do not use properly-defined filters on flow data containing this artifact, this can lead to unexpected results and misconceptions. For example, a filter for flow records with the TCP `ACK`-flag set includes also UDP flows in the case of Exporter 1, 2 and 4. Popular analysis applications, such as *nfdump*,[4] accept these filters without showing any warning to the user. As long as the transport-layer protocol is specified in the filter together with the flags, this artifact will not have any semantic impact on data analysis.

Gaps – In this section we characterize the effects of *flow learn failures* on flow data. This helps to understand whether this artifact is also present in data from other exporters, without having access to any flow cache statistics. Our experiments have shown that the first packets of flows are more likely to be subject to *flow learn failures*, because subsequent packets of accounted flows are matched until the records are expired. Smaller flows are therefore more likely not to be accounted at all, while larger flows may have only their first packets lost. Figure 3.2a shows a time-series of the number of flow records in intervals of 100ms. This data has been collected early in the morning, when Exporter 1 normally starts to run out of capacity. A constant stream of flow records without gaps can be observed until around 7:25, when the number of records increases. Simultaneously, *flow learn failures* (in packets/s) start to be reported by SNMP agents, and several short gaps appear in the time-series. Note that the series are slightly out of phase, because of the higher aggregation of the SNMP measurements.

Interestingly, the gaps caused by *flow learn failures* are periodic, especially when the network load is constantly above the exporter's capacity. When analyzing data from Exporter 1 for two weeks, we have observed that the distribution of the time between gaps is concentrated around multiples of 4s. Furthermore, gaps are not larger than 2s in 95% of the cases. This confirms our assumption about the presence of a cyclic process for expiring records from the flow cache.

[4] `http://nfdump.sourceforge.net`

Gaps can also be caused by other factors, such as the loss of NetFlow packets during their transport from exporters to collectors, or packet loss on the monitored link. Both are typically random events that tend to result either in a homogeneous reduction in the number of flow records, or in non-periodic gaps. Figure 3.2b illustrates the example NetFlow packet loss by showing the time-series of flow records observed at a highly overloaded collector. NetFlow packet sequence number analysis confirms that more than 5% of the NetFlow packets have been lost by the collector during this interval. Several short periods with a reduced number of flow records can be observed, but the series never reaches zero in this example. This demonstrates that gaps in flow data cannot be irrefutably traced back to *flow learn failures*.

We can confirm the existence of gaps in flow data from Exporter 1 and 2. Exporter 3-5 could not be tested, either because they were used in production or because we were not able to exhaust their flow capacity. Exporter 6 handles collisions in software using linked lists and is, therefore, not subject to *flow learn failures*. Under extreme load, it exports flow records earlier, ignoring timeout parameters completely.

Although this artifact has a severe impact on any analysis because of the resulting incomplete data set, we discuss only two examples: anomaly detection and bandwidth estimation. The detection of anomalies (especially flooding attacks) is often based on large sets of small flows. Since the first packets of a flow are especially susceptible to *flow learn failures*, they are more likely to be lost during the flow export process. Anomalies can therefore stay undetected. Besides dropped flow records, peaks in the network traffic may be smoothed due to the load-dependency of the artifact. Since the identification of peaks is essential for bandwidth estimation, traffic analysis may provide invalid estimates.

3.5 Conclusions

In this chapter we have identified, analyzed and quantified five different artifacts in flow data exported by six widely-deployed devices. These artifacts are related to the way in which these devices handle the flow expiration, TCP flags and byte counters, and to imprecisions in the number of exported flow records.

Our analysis shows that the impact of the identified artifacts on the quality of flow data varies, and that in some cases mitigation and recovery procedures can be considered. For example, *non-TCP flow records with TCP `ACK` flag set* can be repaired easily. The *imprecise flow record expiration* artifact can in many cases be ignored if the flow collector aggregates records belonging to the same flow before analysis. However, the remaining artifacts cannot easily be mitigated and they adversely impact the quality of the exported flow data.

The severity of the identified artifacts ultimately depends on their impact on the applications that are using the data. Although analysis applications are usually designed and built to be generic and applicable to any flow data, the

experience gained during our study convinced us that a better way for designing flow-based applications would be to take data artifacts into account. Since the types of artifacts differ from exporter to exporter, we believe that researchers and operators need to be aware of these artifacts to build more robust analysis applications. Moreover, for flow data to be usable for compromise detection, we conclude that dedicated flow export devices (probes) must be used to safeguard the quality of the data.

Part II

Compromise Detection

CHAPTER 4

Compromise Detection for SSH

Although attacks against SSH daemons exist already for years, they are still omnipresent in today's networks and their popularity is every-increasing; In campus networks like the UT's we observe more than 100 large brute-force attacks per day, and many hundreds per day in backbone networks. The fact that SSH is a powerful protocol that allows for remote administration makes that adversaries may use compromised devices for all sorts of malicious activities, such as joining in DDoS attacks and distributing SPAM. Detecting compromises is therefore of utmost importance. In this chapter, we investigate how to detect SSH compromises using flow data. We start by analyzing current SSH attacks to determine their behavior and define an attack state model that is used throughout this thesis. Our approach is based on signatures that allow for detecting phase transitions in several types of brute-force attacks. We start by demonstrating that an elementary approach may result in false positives and negatives due to the presence of network artifacts and specific features of the SSH protocol and related tools. After that, we add two crucial components to our elementary approach. First, we analyze network artifacts and investigate how to improve our detection approach such that it is able to cope with these artifacts. Second, we investigate the effect of including SSH-specific knowledge in our detection algorithm to make it recognize specifics of the SSH protocol and related tools. The main contribution of this chapter are SSH compromise detection algorithms, of which we have validated the correct functioning using our open-source IDS SSHCure. This software was not only used by us, but also by quite a number of companies, governments, CSIRTs and researchers.

The papers related to this chapter are [12], [44], [52], of which [44] received a Best Paper Award at AIMS 2012. Our open-source IDS SSHCure that implements the presented material also received an award, namely the 2015 Communication System Award of the German Association for Computer Science (GI), and contributed to become a co-winner of the Dutch National Cyber Security Research Agenda (NCSRA) in the Ph.D. student competition.

The organization of this chapter is as follows:

- *Section 4.1 motivates the research presented in this chapter and states our contributions.*
- *Section 4.2 describes and explains various types of SSH attacks and their flow-level characteristics.*

- *Section 4.3 presents our elementary detection approach, which aims at identifying attack phase transitions by identifying so-called 'flat' network traffic in flow data.*
- *Section 4.4 presents improvements to the detection of the brute-force phase. We investigate how network artifacts impair our detection and demonstrate how we can improve detection results by enhancing our measurement infrastructure.*
- *Section 4.5 presents our findings as to how we can improve our detection of the compromise phase by including SSH-specific knowledge for identifying compromises.*
- *Section 4.6 concludes this chapter.*

4.1 Background

Brute-force attacks against SSH daemons on the Internet are omnipresent and their popularity is increasing evermore [145]. These attacks, which can be particularly harmful due to a compromise often resulting in full system access, may be fostered by the following facts:

- The number of SSH daemons connected to the Internet is huge. Shodan, for example, reports 26 million connected and scannable daemons in November 2015. In many cases, the actual number of reported daemons will be larger than the number reported, due to system administrators trying to hide the daemon by moving it to another port or firewalling the common port to prevent access from the Internet.
- Although password managers are rapidly gaining popularity and awareness about the use of unique passwords per site or application is improving, people still prefer to use passwords that are relatively easy to remember. These relatively simple passwords can typically be found in *dictionaries*, lists of frequently-used passwords that are used by attackers during brute-force attacks.
- Brute-force attacks against SSH daemons are well-understood and many attack tools are available. These tools are easy to find and simple to use, resulting in powerful tools to be used by a large audience.

Despite the fact that brute-force attacks form a major security threat for many devices connected to the Internet, their mitigation is rather simple. For example, simply prohibiting password-based authentication and switching to key-based authentication would render most attacks useless. Key-based authentication works by storing a public key on a remote device and using a private key to authenticate against it. While it is theoretically possible to try many different keys when key-based authentication is enabled, the number of possible combinations is much larger than with password-based authentication. Another type of mitigation that is typically easier to employ than enforcing key-based authentication is the use of authentication monitors, such as *fail2ban*[1] and *denyhosts*[2]. These tools scan authentication logs for failed authentication attempts and, once a threshold is exceeded, block the source/attacking host.

While mitigation of attacks against SSH daemons appears simple, the alarming number of compromises clearly indicate that the listed mitigation mechanisms are not deployed widely enough. It was only in 2014, for example, when the *Ponemon 2014 SSH Security Vulnerability Report* revealed that 51% of their survey respondents, i.e., Global 2000 companies, admitted to have seen a compromise in the last 24 months [160]. Provided that it is difficult to generally

[1] `http://www.fail2ban.org`
[2] `http://denyhosts.sourceforge.net`

secure all devices connected to the Internet, we take another approach: By deploying an IDS at central observation points in the network, we monitor all SSH daemons without the need of changing any end system. Moreover, by focussing on compromises rather than just attacks, we drastically reduce the number of incident reports to be analyzed by security teams.

4.1.1 Related Work

The core contribution of this thesis is the detection of compromises. By the nature of brute-force attacks, compromises *must* be preceded by (brute-force) authentication attempts. Therefore, reliable detection of compromises requires solid brute-force attack detection algorithms. Three Ph.D. theses and related publications have focussed on the flow-based detection of brute-force attacks before:

- The work in [81] (2010) aims at defining a theoretical model of SSH brute-force attack behavior. It was found that these attacks typically consist of three phases, namely a *scan*, *brute-force* and *compromise* phase. The evaluation of the proposed model was mainly performed on a manual and theoretical basis. Due to the lack of datasets for evaluating the work, the authors have generated their ground-truth datasets in two complementary manners, namely by manually creating the datasets using data collected from honeypots, and by generating datasets in an automated manner using Hidden Markov Models (HMMs). The work is used as the starting point for this thesis and a more extensive description of the three attack phases will be provided in Section 4.2.

- While the work in [81] is completely based on the three brute-force attack phases, the work in [83] (2013) focusses mostly on the *brute-force* phase: Its main contribution is summarized as a "design, implementation and long-term evaluation of flow-based detection of brute-force attacks in high-speed networks". The authors aim at identifying connections belonging to brute-force authentication attempts in flow data by observing connections that are alike in terms of packets, bytes and duration. In this respect, the detection of the *brute-force* phase is very similar to observations of the authors in [81]. Additionally, the authors use a clustering approach for grouping similar connections.

- Similar to the work in [83], the work in [79] (2015) does not specifically target the *compromise* phase. It describes its contribution as focussing on "the detection of dictionary attacks and their distributed variants". The authors investigated various types of SSH brute-force attacks and developed a new attack model to cover those. This allows them to predict, up to a certain extent, which attacks are theoretically possible in the future,

but have not been observed so far 'in the wild'. Based on a survey of existing datasets, the authors use their own platform to construct datasets for validating their work. The actual detection algorithm is based on *sketches*, data structures that enable compact representation of processed data, to identify anomalies in network traffic and conclude the presence of attacks.

We can derive the following conclusions from this enumeration. First, all three works are based on the observation that traffic for authentication attempts, i.e., referred to as traffic in the *brute-force* phase in [81], is alike in terms of packets, bytes and duration. This is not very surprising, given that authentication attempts are identical application-layer events, where only the length of the usernames and password may change. These changes however cause only marginal deviations in network traffic. Second, the detection of compromises is targeted neither by [83], nor by [79]. While both works cite and elaborate on the work presented in [81], the focus on the detection of brute-force attacks and distributed dictionary attacks, respectively. Similarly, even though the ideas of detecting compromises in network traffic were born in [81], they are presented as no more than an observation, without realizing its great potential and advantages above regular attack detection. Third, the work in [81] can be considered a generalization of the works in [83] and [79]. While the latter two mostly target the attack phase that features the authentication attempts, they acknowledge that this phase is often preceded by one or more network scans. Moreover, the work in [83] even uses the observation of a network scan to increase the detection confidence and reduce false positive detections.

4.1.2 Challenges & Contributions

Despite that the three discussed theses have already targeted a similar area of research as the work presented in this thesis, we are targeting several open challenges. The most important challenge is the flow-based detection of compromises, which has not been targeted by any related work before. We therefore define the objective of this chapter as to investigate how to detect devices that are compromised by means of SSH brute-force attacks on the Internet. We use the findings presented in [81] as a starting point for analyzing and describing SSH brute-force attacks in Section 4.2, where we explain our three-phase attack model in detail, as well as modifications we made to the original model presented in [81] based on measurements in various open networks. Based on our SSH attack analysis, we present our detection approach in Section 4.3. This approach aims at identifying attack state transitions and is therefore generic enough to detect attacks against other protocols than SSH as well. We implemented the detection algorithm that was developed for this approach as part of our open-source IDS SSHCure, as a way to demonstrate that flow-based compromise detection is actually feasible and has potential for production usage. Based on many deployments in networks all over the world, we have analyzed false positive and negative detections and

discovered that network artifacts are a major cause of these. We then follow-up on this in two directions:

1. We investigate in Section 4.4 whether enhancements of our measurement infrastructure allow us to overcome any impairments caused by network artifacts.

2. We investigate in Section 4.5 whether the inclusion of domain-specific knowledge on SSH in our detection algorithm allows us to be more resilient against false detection caused by peculiarities of the SSH protocol. Additionally, we expect this to make our detection algorithm more resilient against network artifacts as well, since we do not fully depend on attack state transitions anymore.

Finally, we close this chapter in Section 4.6, where we draw our conclusions.

4.2 SSH Attack Analysis

Brute-force attacks aim at compromising user accounts by trying many combinations of usernames and passwords. One particular type of brute-force attack is the dictionary attack. Attacks using dictionaries are particularly effective, due to the fact that dictionaries feature the most common usernames and passwords. Also, purely random passwords, which are typically not found in dictionaries, are difficult to remember and therefore less common than passwords that are simple and easy to remember. A recent study on SSH brute-force attacks has shown that the vast majority of attacks is fully automated using software tools and that attackers rely on heavily shared dictionaries [26].

Before targets can be attacked, they must be discovered. This can be done by scanning a network, e.g., using tools like `nmap`,[3] or by using publicly available lists of potential targets, such as those provided on PasteBin,[4] or gathered by services like Censys.io[5] and Shodan.[6] Once potential targets have been discovered and selected, brute-force attacks can be launched. Eventually, targets may be compromised, depending on whether a valid pair of credentials was used. To describe these multi-phase attacks, we define the following attack phases:

- ***Scan* phase** – An attacker performs horizontal scans over a network to find targets, i.e., active daemons on a particular port. In the case of SSH daemons, the scanned port is usually TCP port 22. A list of attackable targets can of course also be obtained using scanning services, as explained before.

[3] `https://nmap.org`
[4] `https://pastebin.com`
[5] `https://censys.io`
[6] `https://shodan.io`

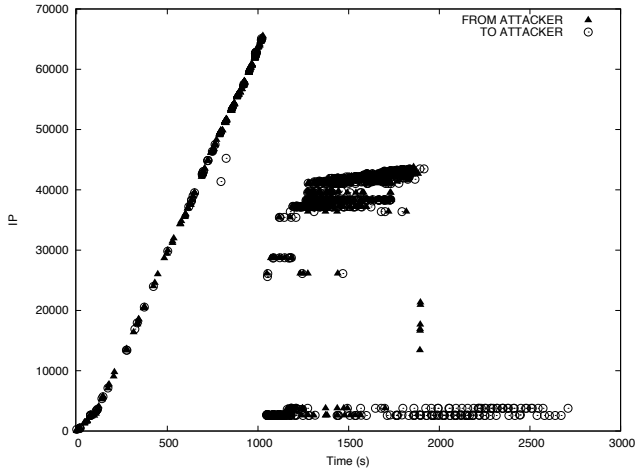


Figure 4.1: Time-series of SSH attack over IP address space, from [68].

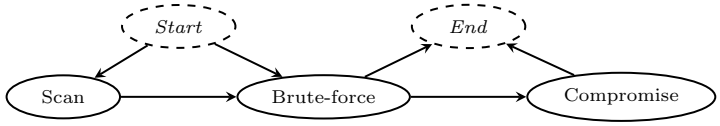


Figure 4.2: Brute-force attack phase transitions.

- ***Brute-force* phase** – An attacker performs many authentication attempts on target hosts, using a large number of username and password combinations, either based on dictionaries or using randomly generated passwords.
- ***Compromise* phase**[7] – An attacker has gained access to a target host by using a correct pair of credentials. The host may either be actively misused right-away or left aside for misuse at a later point in time.

To avoid any confusion between the denotation of the nature of an attack (i.e., brute-force attack) and attack phases (e.g., *brute-force* phase), we consistently use italics to denote attack phases throughout this thesis.

A typical SSH brute-force attack, monitored at the campus network of the UT, is visualized in Figure 4.1. The figure shows how an attacker connects to most IP addresses in the UT's class B IPv4 address block, consisting of 2^{16} IPv4 addresses. Every point in the plot corresponds to a flow record from the attacker to the campus network, or to a flow records from campus hosts back to the attacker. For the specific case depicted in Figure 4.1, the *scan* phase takes place

[7]We use the more intuitive *compromise* phase to denote the phase that is named *die-off* phase in [68].

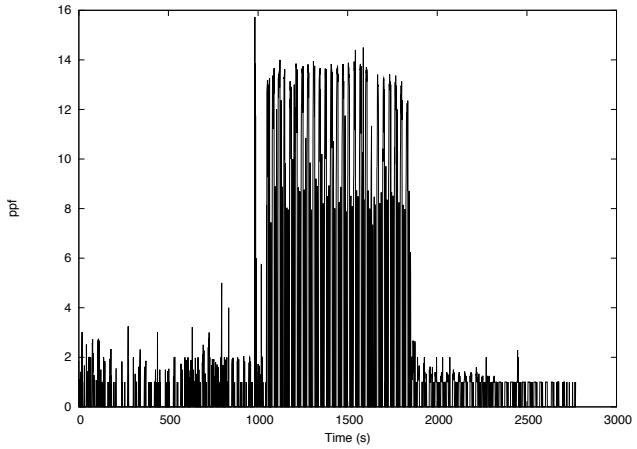


Figure 4.3: Time-series of SSH attack in terms of Packets per flow (PPF), from [68].

from the beginning of the attack until $t \approx 1000$, immediately followed by the *brute-force* phase, which terminates at $t \approx 1750$. Finally, residual traffic, i.e., the *compromise* phase, is present on the network until $t \approx 2750$ after the beginning of the attack.

All supported SSH attack state transitions are shown in Figure 4.2. Although these attack phases feel rather natural, they were formalized for the first time in the context of network flows only in 2009 in [68], where it was found that the aforementioned attack phases could be easily identified in flow data by the number of PPF, as shown in Figure 4.3. Here, the *scan* phase features a very low number of packets per flow, since it merely consists of one or two TCP `SYN` packets. The *brute-force* phase, however, is characterized by a significantly higher number of PPF, due to the SSH connection initiation and one or more authentication attempts. In literature, traffic in the *brute-force* phase is often described as being *flat*, because repeated application-layer actions/events (e.g., authentication attempts) result in connections that are alike in terms of packets, bytes and duration. In case of a compromise, we may either observe a number of PPF that is higher than the *brute-force* phase's in case the target is being actively misused, or a lower, in case the connection to the target is maintained but idle.

We consider the work in [68] as one of the cornerstones of this thesis, but use its findings and contributions with two modifications. First, attacks can also start in the *brute-force* phase for the following reasons, because (a) the *scan* phase can have taken place in the past, e.g., before traffic observation has started, and (b) our investigation of attack tools (Section 4.5) has shown that attacks can start directly from the *brute-force* phase, as scans can have been performed by another

host or the target may be known in advance by an attacker. Second, rather than the suggested range $r = [10, 15]$ for *brute-force* phase traffic, we use the range $r = [11, 51]$. Here, 11 is the minimum number of packets needed for a single authentication attempt, and 51 the highest number of PPF observed for brute-force phase traffic. These numbers are backed up by measurements and higher than reported by related works, which report maximum values of around 30 [75], [83]. We have found that Cisco appliances and Mac OS X are the main cause of these high values.

Although our attack analysis presented is presented here in the context of SSH, it is very similar for other services for which attacks look similar on the network-level [83]. For a service to correspond with the presented analysis, it should feature some sort of authentication mechanism that is *required* to advance in the protocol state, similar to SSH. Exemplary services are Remote Desktop Protocol (RDP) and Virtual Network Computing (VNC), but we consider these services out of scope of this thesis.

4.3 Detecting SSH Brute-force Attacks

In the previous section we have analyzed brute-force attacks against SSH daemons and shown that they typically consist of three phases. Based on this attack phase model, visualized in Figure 4.2, we present in this section our first approach to detecting SSH attacks, aiming at the identification of the three attack phases in network traffic.

4.3.1 Algorithm

The objective of our detection algorithm is to classify attacks into one or more subsequent attack phases, according to our brute-force attack model. In the remainder of this section, we present how the algorithm detects each attack phase. All presented algorithm parameters based on traffic metrics are calculated with a time granularity of 1 minute, unless indicated differently.

The detection of network scans and brute-force attacks was already discussed by multiple other works, such as [79], [83]. In this section, we use very similar algorithms to those described in literature, which differ only in terms of implementation details and thresholds. All thresholds described here are subject to change if deployed in networks of a different nature than the campus network of the UT.

Scan phase

During the *scan* phase, an attacker probes for the presence of specific services on one or more hosts in a network, which results in several clearly distinguishable characteristics. Since attacks are assumed to originate from a single attacker,

many small flows from the attacking host to a large number of targets is an indication of an attack in the *scan* phase. The detection algorithm selects suspicious traffic by means of the following two metrics:

- **PPF** – The algorithm uses an upper limit of 2 PPF during the *scan* phase, which is a typical feature of a port scan. Unlike attack traffic, regular SSH connections will use the same TCP connection once established, which results in a higher value for the PPF metric.
- **Minimum number of flow records** – If only the PPF metric was used for distinguishing *scan* traffic, regular SSH traffic could still be classified erroneously as *scan* traffic. This is for example the case with SSH sessions involving sporadic activity, because flow monitoring devices export long-lived flows using multiple flow records (due to the use of timeouts). This is not a limitation of our algorithm, but inherent to the design of flow export devices and technologies. To overcome this deficiency, we define a threshold for the number of flow records per attack in the *scan* phase, N_s. Based on measurements in the campus network of the UT in 2011, $N_s = 200$ flow records per time interval of 1 minute was chosen. This threshold corresponds to roughly 200 SSH connection attempts per minute, which is almost guaranteed not to be regular SSH usage.

Brute-force phase

The detection of the *brute-force* phase is typically performed by comparing the characteristics of two or more flow records to identify possible attacks. During the *brute-force* phase, a high-intensity brute-force attack is performed on one or more targets on which a service is found active. The *brute-force* phase typically contains many flow records with an equal number of PPF, since flows featuring an equivalent number of login attempts between the same client and server typically consist of the same number of packets. Should a compromise ensue, the *compromise* phase is reached.

The *brute-force* phase is the only phase where flat traffic should be predominant. The concept of an equal number of PPF, i.e., flat traffic, for brute-force attack detection forms the basis of our *brute-force* phase detection algorithm, which considers the number of PPF in consecutive flow records. The algorithm starts with a preselection of source and destination IP address pairs for which flow records have a PPF value of $x \in [11, 51]$. For each of these preselected address pairs, the most frequently used PPF value is taken as the baseline for determining brute-force behavior. This baseline is then used for comparing consecutive flow records with identical PPF values to. If at least N_b consecutive flows feature the baseline number of PPF, a brute-force attack is recognized. We set the threshold N_b to five[8], and the result of the detection algorithm is a list of attacks. In

[8]Note that five consecutive flow records with the same number of PPF would represent 15 failed login attempts in a benign situation, which we consider highly unlikely.

the remainder of this work, we define an attack as a set of one or more targets featuring brute-force behavior for a given attacker, i.e., where every target in the set has reached N. A tuple is defined as a pair of attacker and target, such that every attack consists of one or more tuples.

If more than N_b flow records feature the same number of PPF, a brute-force attack is detected. On the one hand, false negatives, i.e., undetected attacks, can occur in this context when brute-force attack flows end up with diverse PPF values, causing the threshold N_b to not be reached, even though the application-layer activity remains similar. On the other hand, false positives, i.e., false alarms, can occur when non-attack flows end up with equal PPF values, enough to reach the threshold N.

For increased confidence, we may combine the detection results of the *scan* phase with results of the *brute-force* phase; It is more likely that a detection in the *brute-force* phase is indeed an attack if it was preceded by a network scan, than if it was not preceded by a scan.

Compromise phase

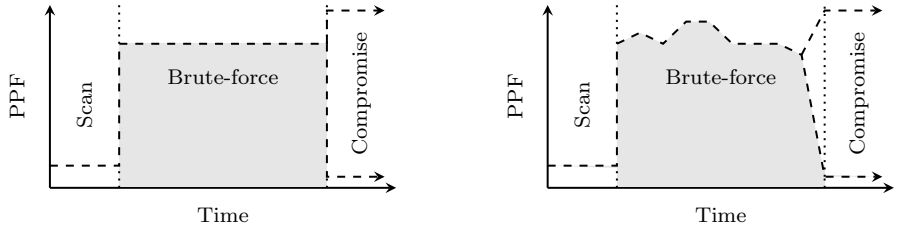
Attacks can only progress to the *compromise* phase after passing through the *brute-force* phase. The *compromise* phase can again be identified using the PPF metric, which should change significantly on transition from the *brute-force* phase. It should be noted that the change can either be positive, i.e., resulting in a higher number of packets, in case the compromised target is being actively misused, or negative, i.e., resulting in a lower number of packets, in case the target is (temporarily) left aside. The threshold for classifying a measured number of PPF as a compromise depends on the baseline used for the detection of the *brute-force* phase. In general, we can consider any deviating number of PPF a compromise, although one may use a safety margin to reduce the number of false positives.

4.3.2 Prototype

The work in [68] aimed at defining a theoretical model of a brute-force attack behavior, including the *scan*, *brute-force* and *compromise* phase. To illustrate that flow-based compromise detection is actually feasible, we developed an open-source IDS named *SSHCure*.[9] *SSHCure* has a strong focus on detecting the three attack phases of SSH brute-force attacks and was the first (flow-based) IDS that could report on compromises. Since our goal was to reach a wide audience, we developed *SSHCure* as part of the popular flow collection software *NfSen*[10] and managed to attract quite some attention to the development by visiting conferences, performing demos for companies, etc. Although we cannot name individual users of *SSHCure* for the sake of privacy, we are aware of a wide

[9] To be pronounced as *she-cure*. The latest version is available at `https://github.com/sshcure/sshcure`

[10] `http://nfsen.sourceforge.net`



(a) Theory (b) Practice

Figure 4.4: Brute-force attack behavior.

user base, ranging from small Web-hosting companies to backbone networks and governmental CSIRTs.

4.3.3 Lessons Learned

Since *SSHCure* relies on flow data exported using NetFlow or IPFIX, it was believed to work on flow data from any source, like any other flow data analysis software. However, as soon as *SSHCure* was released to the public and we requested feedback from its users, we received mixed results. For some, *SSHCure* worked just great and provided them with a powerful tool for detecting compromises. Others, however, were reporting on false positive and negative detections, i.e., real attacks that stayed under the radar, and reported incidents that were actually no attacks, respectively.

Analysis of the reported issues revealed that a large number of problems was caused by a broken assumption; Although flat traffic was thought to be significative of traffic in the *brute-force* phase [79], [83], we discovered this to often not be the case. So instead of the theoretical attack behavior that is depicted in Figure 4.4a, we discovered that many attacks feature deviations in the number of PPF, as shown in Figure 4.4b for network and measurement artifacts. Once false detections are present in the *brute-force* phase, they propagate to the *compromise* phase as well, as a direct consequence of our three-phase attack model. We are also aware of attack obfuscation techniques that aim at generating network-level behavior similar to what is shown in Figure 4.4b. However, we consider attack obfuscation techniques to be out of the scope of this thesis, since (a) these techniques have been discussed in other works already, (b) these techniques will always be a 'cat-and-mouse game' between attackers and defenders, and (c) measurements have shown that the vast majority of attacks do not use obfuscation techniques [12].

Another class of issues that we found to cause false positive and negatives in our detection of the *compromise* phase were specific characteristics of the SSH protocol and related tools. For example, attack mitigation tools that scan log files and block hosts after more than a selected number of failed authentication

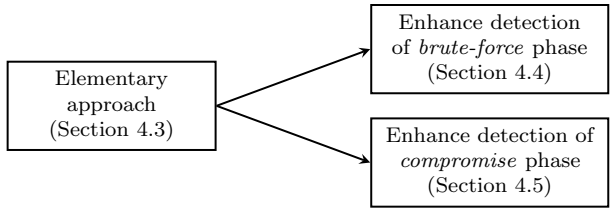


Figure 4.5: Directions of research presented in remainder of this chapter.

attempts, cause traffic in the *brute-force* phase to all of a sudden feature a significantly different behavior, making our compromise detection approach believe to be dealing with a compromise. Also, we found that popular SSH daemons are shipped with rate-limiting features, for example, that also affect the flatness of the traffic in the *brute-force* phase, resulting in false positive detections of compromises.

In the remainder of this chapter, we present two directions of research for improving our elementary detection approach presented in this section, as shown in Figure 4.5. First, we investigate in Section 4.4 how network artifacts affect the flatness of attack traffic and therefore eventually impair our detection of compromises. The main problem is that we cannot discriminate TCP phenomena like retransmissions and control information in flow data. We therefore aim at enhancing our flow measurements such that network artifacts like TCP retransmissions and control information can be identified. Second, we enhance our detection algorithm in Section 4.5, by including SSH-specific knowledge to be more resilient against network artifacts and specifics of popular related tools, such as attack mitigation systems. Although the resulting detection algorithm is not generic anymore to a whole class of protocols that requires successful authentication to advance in the protocol's state machine, it allows us to identify SSH protocol behavior that would otherwise be incorrectly classified as a compromise.

4.4 Analysis of Network Traffic Flatness

Many types of brute-force attacks are known to exhibit a characteristic flat behavior at the network-level, meaning that connections belonging to an attack feature a similar number of packets and bytes, and duration. Flat traffic usually results from repeating similar application-layer actions, such as login attempts in a brute-force attack. For typical attacks, hundreds of attempts span over multiple connections, with each connection containing the same, small number of attempts. The characteristic flat behavior is used by many IDSs, both for identifying the presence of attacks and – once detected – for observing deviations, pointing out potential compromises, for example. However, flatness of network traffic may become indistinct when TCP retransmissions and control information

come into play. These TCP phenomena affect not only intrusion detection, but also other forms of network traffic analysis.

In this chapter, we analyze the impact of retransmissions and control information on network traffic based on traffic measurements. To do so, we have developed a flow exporter extension that was deployed in both a campus and a backbone network. Also, we show that intrusion detection results (based on the algorithm presented in Section 4.3) improve dramatically by up to 16 percentage points once IDSs are able to 'flatten' network traffic again, which we have validated by means of analyzing log files of almost 60 hosts over a period of one month.

4.4.1 Background

To facilitate reliable data delivery between endpoints, TCP uses a cumulative acknowledgement scheme in which sequence and acknowledgement numbers are used to signal the reception of data. In the absence of any feedback from the data receiver, a Retransmission TimeOut (RTO) is used to ensure delivery, which is based on the estimated Smoothed Round-Trip Time (SRTT). Due to unexpected delays or reordering of packets in the network, retransmissions can occur spuriously. For example, when a packet or its acknowledgement is delayed unexpectedly rather than lost, the RTO timer expires and the packet is retransmitted. Also, a fast retransmission may be sent when a certain number of consecutive duplicate acknowledgements is received, signalling the potential loss of packets to the sender. Due to reordering of packets, duplicate acknowledgements may be sent even though no packet has gotten lost. These duplicate acknowledgements can trigger a spurious fast retransmission. In both examples, spurious retransmissions and their duplicate acknowledgements cause additional packets and bytes in network traffic and hence affect the potential flatness of a connection.

To optimize network throughput while avoiding congestion or overloading an endpoint, TCP uses several techniques, such as flow control, based on a sliding window, and the *delayed ACK* mechanism. To realize flow control, the *receive window* needs to be signalled from receiver to sender, and under the *delayed ACK* mechanism, data acknowledgements are held back for a brief delay to save overhead. If data or additional control information becomes available during the delay, the held back acknowledgement can be combined with this information. For some forms of control information, such as data acknowledgements and *receive window* changes, the *delayed ACK* mechanism and circumstances dictate whether a dedicated packet is sent to carry the control information to the endpoint. For example, if during a *delayed ACK* data is pushed down from the application-layer, the held back acknowledgement can be *piggybacked* with a data packet. This prevents sending a dedicated acknowledgement with no payload. Also, the *delayed ACK* mechanism allows for the cumulative acknowledgement of two data packets received in rapid success. This too saves sending a dedicated packet. If the *receive window* changes at the receiver, this information can be combined

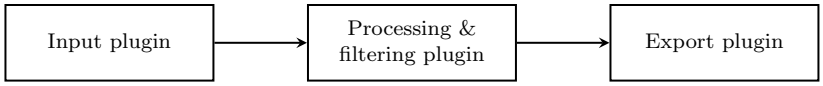


Figure 4.6: INVEA-TECH FlowMon platform architecture, as of FlowMon Probe 6.x.

with a held back acknowledgement, again saving a dedicated packet. Sometimes, however, the *receive window* expands when there is no data to acknowledge, in which case a dedicated *window update* needs to be sent. Whether or not such dedicated packets are sent affect the flatness of network traffic.

There are also types of control information that are always sent in separate packets: *Zero Window probes* and responses, *KeepAlive Probes* and responses, and `RST` packets. Also, depending on the TCP implementation, a three-way `FIN` close sequence may not be supported, thereby potentially introducing an additional packet during the connection termination. Any of these additional packets obviously affect the flatness of network traffic as well.

It is important to note that the presence of the aforementioned situations mostly depends on network conditions, resource availability and scheduling on endpoints, whether or not there is data to send or acknowledge, and timing.

4.4.2 Implementation

To export information that allows for the discrimination of TCP retransmissions and control information in flow data, several IPFIX IEs were defined and implemented as part of a flow Metering Process. This section describes these IEs and the accompanying implementation.

We have defined IPFIX IEs for each of the TCP protocol phenomena discussed (in italics) in Section 4.4.1. To facilitate the export of these IEs, we have developed an extension to INVEA-TECH's FlowMon flow exporter. This platform was chosen because of its highly customizable plugin architecture, and because we have full control over it in our networks. The complete architecture is shown in Figure 4.6. It is based on plugins for data input, flow record processing & filtering, and export. Input plugins process data from a given source, such as one or more line cards, and are responsible for creating flow cache entries, one entry per active flow. Process plugins allow for the manipulation of these cache entries once these entries have been created. The process plugin type is best suited for program logic that does not necessarily require a packet's payload anymore. The export plugin is responsible for exporting cache entries by sending flow records to a collector using NetFlow or IPFIX. From within these plugin types, actions can be hooked to events such as flow entries being added to, updated in or expired from the flow cache. Among these actions is the filtering of flow cache entries to prevent them from being exported.

Dataset	Packets	Bytes	Flows	Retransmissions		Ctrl. information	
				Packets	Bytes	Packets	Bytes
UT	370.73 G	291.64 TiB	7.35 G	5.30 G (1.43%)	2.83 TiB (0.97%)	100.50 G (27.11%)	4.30 TiB (1.47%)
CESNET	257.38 G	227.67 TiB	3.57 G	8.29 G (3.22%)	2.78 TiB (1.22%)	83.61 G (32.48%)	3.48 TiB (1.53%)

Table 4.1: Datasets and their characteristics.

Our extension comes in the form of an input plugin, because it evaluates the payload of packets. The plugin measures TCP retransmissions and control information packets, and stores and maintains related counters in the flow cache. To recognize these particular packets, TCP conversations are analyzed in real-time by evaluating sequence and acknowledgement numbers, timestamps, flags, receive window sizes, and payload sizes. This implementation is heavily based on the TCP packet dissector used by *Wireshark*.[11]

For the TCP analysis to be accurate, it is crucial that packets in both directions of a TCP conversation pass through the observation point. Otherwise, the housekeeping of sequence and acknowledgement numbers may be affected, which obviously impairs the analysis. The same is true when packets are lost downstream of the observation point. We are also aware of the fact that the TCP packet dissector used by *Wireshark* cannot but misclassify packets in its on-the-fly analysis in some cases, especially when packets are reordered. To optimize our plugins to work on high-speed links, e.g., of *10 Gbps* and higher, we accept these exceptional cases for the sake of performance.

4.4.3 Measuring TCP Retransmissions & Control Information

Our first step towards understanding the impact of TCP retransmissions and control information is to measure it in two networks that are different in nature. Two datasets were collected, as shown in Table 4.1, consisting of only TCP flow data. Dataset *UT* was collected on the campus network of the UT in the course of July/August 2014 (31 days). This network features a publicly routable /16 network address block with connections to faculty buildings, student and staff residences, etc. Due to the residential aspect of the campus network it also routes private c.q. non-academic Internet traffic. Furthermore, the campus network houses mirror servers for popular open-source software, such as Ubuntu. Dataset *CESNET* was collected on a backbone link of the Czech NREN, specifically the link between CESNET and the 'commercial Internet'. The dataset was collected in the course of August/September 2014 (31 days).Due to the academic

[11] http://www.wireshark.org/

Dataset	Retransmissions		Fast Retransmissions	
	Packets	Bytes	Packets	Bytes
UT	95.50%	89.54%	4.50%	10.46%
CESNET	97.87%	91.27%	2.13%	8.73%

Table 4.2: Distribution of retransmitted packets and bytes.

nature of these networks, the relative amount of traffic during summer holidays is considerably lower than during working days.

The remainder of this section is organized in two parts. We start by analyzing retransmissions and control information in detail based on our measurements. After that, we perform a similar analysis only for SSH traffic, given that the validation of this work will be performed in the context of SSH intrusion detection.

4.4.3.1 Overall Traffic

Details on the number of retransmitted packets and bytes, and the amount of control information in terms of packets and bytes are shown in Table 4.1. Several observations can be made. On the one hand, TCP control information is mostly visible in terms of packets. On the other hand, retransmissions contribute more towards the percentage of bytes. Another observation is that there are many more packets with control information than there are retransmitted packets. This is mainly because many control information packet types, such as those that result from the *delayed ACK* mechanism, are sent under all network conditions, while retransmissions appear more frequently during network congestion, for example.

The distribution of retransmission types in terms of packets and bytes is shown in Table 4.2. As can be observed, most retransmissions are regular retransmissions. Also, for each dataset, the fraction of the total number of bytes for the fast retransmission type is higher than the packet fraction. We believe this is because regular retransmissions can also contain no payload, e.g., retransmissions of empty TCP `SYN` and `FIN` segments bring down the average number of bytes per retransmitted packet. Considering the *UT* dataset, it shows that while 4.50% of retransmitted packets are of the fast type, these do account for 10.46% of the number of retransmitted bytes. For the *CESNET* dataset, these numbers are 2.13% and 8.73%, respectively.

The number of retransmitted packets and fast retransmitted packets within every five-minute interval in the 31 days of the *UT* dataset is shown in Figure 4.7. A diurnal pattern can be clearly identified, which follows the working hours at faculty buildings, and the presence of on-campus residents. While Table 4.1 provides absolute numbers, and as such is not specific about the points in time at which events occur, Figure 4.7 shows that retransmissions occur at any time of the day. The two outlying groups of retransmitted packets around *5 Aug 18:00* and *10 Aug 18:00* coincide with severe SSH dictionary attacks from China

Type	Dataset	
	UT	CESNET
Duplicate ACK	5.24%	1.77%
Non-piggybacked ACK	7.61%	11.71%
Consecutive empty ACK	83.13%	80.60%
Window Update	2.02%	1.88%
Zero Window Probe (ZWP)	< 0.01%	0.01%
ZWP response	< 0.01%	< 0.01%
`RST`	0.87%	2.59%
Four-way close packet	0.10%	0.21%
KeepAlive Probe	0.54%	0.74%
KeepAlive Response	0.48%	0.48%

Table 4.3: Distribution of control information packets.

Dataset	Retransmissions		Control Information	
	Packets	Bytes	Packets	Bytes
UT	1488.18 M (9.53%)	167.36 GiB (1.45%)	3269.24 M (20.93%)	145.19 GiB (1.26%)
CESNET	153.54 M (2.10%)	25.44 GiB (1.54%)	1767.31 M (24.15%)	76.78 GiB (4.64%)

Table 4.4: TCP Retransmissions & Control Information for SSH data subset.

that involve many retransmissions, which makes these anomalies visible in our measurements. These attacks will be discussed later as part of our validation in Section 4.4.5.

The distribution of the various types of control information packets is shown in Table 4.3. As can be seen, packets related to the *delayed ACK* mechanism, i.e., *non-piggybacked ACKs* and *consecutive empty ACKs*, account for large percentages of the total number of control information packets in each dataset. For example, *non-piggybacked ACKs* take up 7.61% and 11.71% in *UT* and *CESNET*, respectively. Another example is the *consecutive empty ACK*, with 83.13% in *UT* and 80.60% in *CESNET*.

Given the significant presence of TCP retransmissions and control information in our measurements in two networks that are different in nature, we conclude that these packets are omnipresent on the Internet. Also, we believe to have demonstrated that the flatness of originally flat network traffic on the Internet is likely affected by this omnipresence, as theorized in Section 4.4.1.

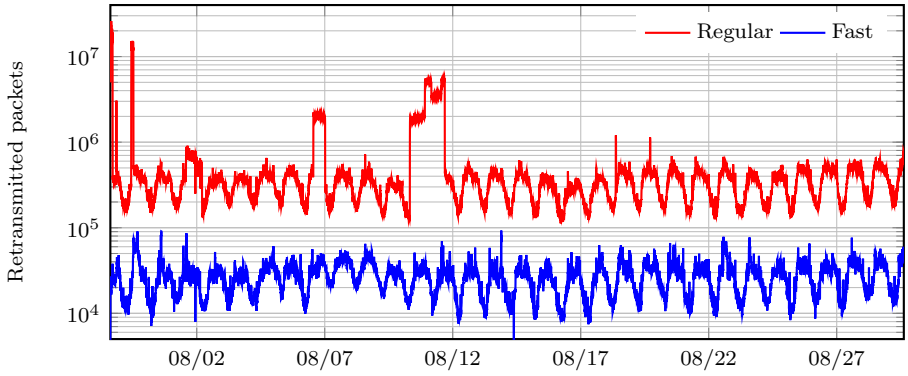


Figure 4.7: Retransmissions over time.

Dataset	Retransmissions		Fast Retransmissions	
	Packets	Bytes	Packets	Bytes
UT	99.71%	96.47%	0.29%	3.53%
CESNET	99.87%	99.07%	0.13%	0.93%

Table 4.5: Distribution of retransmitted packets and bytes for SSH data subset.

4.4.4 SSH Traffic

The SSH traffic considered in this work was obtained by filtering the datasets presented in Table 4.1 for traffic on port 22, yielding 11.29 TiB of traffic for *UT* and 1.62 TiB for *CESNET*. Details on the number of retransmissions and control information packets and bytes are shown in Table 4.4. Several observations can be made when comparing the SSH traffic to the overall traffic. First, for *CESNET*, the relative percentage of retransmissions is lower in the SSH-only traffic than in the overall traffic, at 2.10% versus 3.22%. For *UT*, however, it is much higher, namely 9.53% versus 1.43%. This is because the *UT* dataset contains several large-scale SSH attacks, as discussed previously alongside Figure 4.7. Second, control information in the SSH datasets is more dominant than retransmissions in terms of packets and bytes, which is similar in the overall traffic. Third, considering that the overall traffic in *UT* is only 50% larger than in *CESNET* in terms of bytes (from Table 4.1), the relative amount of SSH traffic in *UT* is much larger than in *CESNET*.

As for retransmissions in SSH traffic, the distribution of these in terms of packets and bytes is shown in Table 4.5. Compared to the distribution of retransmissions in the overall traffic, it can be observed that a higher percentage in the SSH traffic is of the regular retransmission type. In the *UT* dataset, only 0.29% of retransmissions are classified as fast retransmissions, in contrast to a figure of 4.50% in the respective overall traffic. For *CESNET*, these numbers are

Type	Dataset	
	UT	CESNET
Duplicate ACK	1.87%	2.70%
Non-piggybacked ACK	7.76%	63.63%
Consecutive empty ACK	89.58%	25.73%
Window Update	0.37%	0.35%
Zero Window probe	0.00%	$< 0.01\%$
ZWP response	0.00%	$< 0.01\%$
`RST`	0.31%	5.43%
Four-way close packet	0.09%	1.52%
KeepAlive probe	0.00%	0.35%
KeepAlive response	0.00%	0.31%

Table 4.6: Distribution of control information packets for SSH data subset.

0.13% and 2.13%. Relative differences between the overall traffic and the respective SSH-only traffic thus follow the same trend. We believe that this is the case because it is less common for SSH connections to have four or more consecutive packets with payload sent by one endpoint within a short period. In other words, there are not enough consecutive data packets to trigger a fast retransmission.

The distribution of control information in SSH traffic is shown in Table 4.6. This distribution features several key differences compared to the full datasets (see Table 4.3). A prime example is the significantly lower number of packets related to *KeepAlive*, especially for *UT* where there are none at all. A possible explanation for this is that the majority of SSH connections is short-lived, or otherwise active enough to not trigger the TCP *KeepAlive* timer, which is typically in the order of hours [87]. Another observation is that while the distribution of control information types is very similar within the full datasets collected on different networks, this is not the case anymore for the SSH datasets. For example, in the *UT* dataset, 0.30% of all SSH packets are *RSTs*, while *RSTs* account for 5.43% in *CESNET*. We believe that this is due to increased scanning activity. Also, for *CESNET*, *non-piggybacked ACKs* are at a staggering 63.63%, whereas in *UT* they account for only 7.76%. We believe these differences stem from the fact that a lot more data is sent within SSH connections on the UT network. Furthermore, for the *UT* and *CESNET* datasets it can be seen that *four-way close* features only very small percentages of control information packets, namely 0.09% and 1.52%, respectively. This leads us to believe that SSH network traffic is typically not affected much by this type of control information.

4.4.5 Validation

In this section, we quantify and study the effects of TCP retransmissions and control information on flow analysis applications, in the context of flow-based SSH intrusion detection. The validation methodology is discussed in Section 4.4.5.1. Finally, in Section 4.4.5.2, we present the validation results.

4.4.5.1 Methodology

We perform the validation of this work by executing the (state-of-the-art) detection algorithm presented in Section 4.3 on the datasets listed in Section 4.1. Instead of only considering the regular number of PPF in the detection algorithm, as would be the case in a regular flow monitoring setup, we also consider a compensated number of PPF. The compensated number of PPF consists of the number of the total number of packets metered for each flow minus the number of packets of all retransmissions and TCP control information fields. Ultimately, this should result in flat traffic when it comes to attacks.

By comparing the detection results when using non-compensated and compensated data, we can quantitatively evaluate the gain of 'flattening' traffic in the context of SSH intrusion detection. We perform the comparison in two dimensions – attacks and tuples – as this allows us to discover potential differences in the impact of compensation. Although comparing the number of detections in terms of attacks and tuples before and after compensation provides an indication of the detection improvements, it does not reveal anything about to accuracy of these detection outcomes. To assess these accuracies, we have performed a large-scale validation by collecting authentication logs of 58 hosts on the campus network of the UT – 56 servers and 2 honeypots – to serve as the ground-truth for validation. These authentication logs are the only means of validating whether a host has really been under attack. Since we only have the logs for UT hosts, we only consider the *UT* dataset in this part of the validation.

In the authentication logs, a minimum number of failed attempts must be encountered for the behavior to be considered a dictionary attack. Since the detection algorithm considers at least N consecutive flow records, only N or more connections to the SSH server that contain at least one failed attempt are considered. This comes down to at least five sessions with one or more authentication failures each. By evaluating log entries featuring this property, we created a list of attacks to serve as ground-truth for validation. This ground-truth can then be used for expressing the accuracy of the algorithm, both in terms of attacks and tuples, by comparing detection results to the ground-truth based on the following metrics:

- *True Positives* (TP) – Attacks/tuples correctly classified to feature a *brute-force* phase, for which 5 or more sessions with authentication failures are reported in the ground-truth.

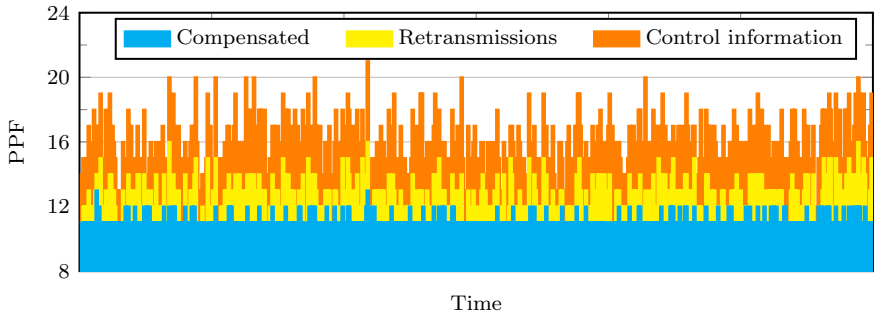


Figure 4.8: Compensated number of PPF in brute-force flow records.

- *False Positive* (FP) – Attacks/tuples incorrectly classified to feature a *brute-force* phase, for which less than 5 sessions with authentication failures are reported in the ground-truth.
- *True Negatives* (TN) – Attacks/tuples correctly classified to not feature a *brute-force* phase, for which less than 5 sessions with authentication failures are reported in the ground-truth.
- *False Negatives* (FN) – Attacks/tuples incorrectly classified to not feature a *brute-force* phase, for which 5 or more sessions with authentication failures are reported in the ground-truth.

Using these metrics, we can evaluate the differences in the detection algorithm for the non-compensated and compensated cases in terms of accuracy (*Acc*), which is defined as follows:

$$Acc = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}} \tag{4.1}$$

In addition, to understand the relation between TCP control information and retransmissions, and geographical locations, we determine the physical origin of attacks and tuples based on a snapshot of the MaxMind GeoIP[12] database at the time of the measurements. The physical location can reveal why certain attacks or the majority of tuples are more likely to be detected only after compensation, as we hypothesize that retransmissions are strongly bound to the geographical distance between attackers and targets.

4.4.5.2 Results

The best way to visualize the achievements of this work is by means of a plot, as shown in Figure 4.8. This figure shows the traffic in terms of the number of PPF

[12]We have used MaxMind's *GeoLite City* database, which can be retrieved from `http://dev.maxmind.com/geoip/legacy/geolite/`.

Dataset	Country	Non-compensated	Compensated
UT	China	1817	2347 (+29%)
	Netherlands	317	694 (+119%)
	Venezuela	195	233 (+19%)
	Russian Federation	165	189 (+15%)
	Chile	154	164 (+6%)
	Other	851	1080 (+27%)
	Total	**3499**	**4707 (+35%)**
CESNET	China	15239	19683 (+29%)
	United States	316	(+14%)
	Brazil	239	257 (+8%)
	Korea	146	170 (+17%)
	Turkey	124	139 (+12%)
	Other	1075	1420 (+32%)
	Total	**17139**	**21985 (+28%)**

Table 4.7: Top five attack origins in terms of attacks.

over time between a single tuple of attacker and target. Clearly, the original network traffic (i.e., the sum of the three series in the figure) is not flat, but after compensating for control information packets and retransmissions, traffic that is almost flat remains. Occasional variations in the remaining number of PPF after compensation are the result of the performance trade-off discussed in Section 4.4.2. We accept these variations, considering that most attacks feature a large enough number of flows to reach the threshold N.

The results of operating the detection algorithm on the considered datasets, both with and without PPF compensation, are shown in Table 4.7 for attacks and Table 4.8 for tuples. The number of detected attacks and tuples is considerably higher after compensation for both datasets. In *CESNET*, the total number of detected attacks is about a fourth times higher after compensation, i.e., from 9475 to 11849, while the improvement in terms of tuples is at 40%. For the *UT* dataset, a gain of 35% in terms of attacks can be observed – from 3499 to 4707 – and a gain of 45% in terms of tuples. The reason for the improvement in terms of attacks is that without compensation, the effects of retransmissions and control information hinder the detection for all tuples of an attack and, as such, the corresponding attack itself is also not detected.

Since we assume that retransmissions depend in part on the geographical location of and route between attacker and target, we show for each dataset the five countries from which most attacks originate, both in terms of attacks (Table 4.7) and tuples (Table 4.8). The total number of countries involved in attacks is 60 for the *UT* dataset, and 71 for *CESNET*. Furthermore, we show the number of attacks and tuples reported only after compensation for those

Dataset	Country	Non-compensated	Compensated
UT	China	31887	55218 (+73%)
	Netherlands	11048	11646 (+5%)
	United States	3573	4203 (+18%)
	Vietnam	2358	2396 (+2%)
	Germany	1592	1642 (+3%)
	Other	10197	12939 (+27%)
	Total	**60655**	**88044 (+45%)**
CESNET	China	799840	1109458 (+39%)
	United States	37161	41230 (+11%)
	France	16096	22818 (+42%)
	Korea	10051	10890 (+8%)
	Malaysia	5579	5811 (+4%)
	Other	36994	48521 (+31%)
	Total	**905721**	**1234659 (+36%)**

Table 4.8: Top five attack origins in terms of tuples.

countries. Several observations can be made from the results. First, regarding the *UT* dataset, many attacks that are detected only after compensation have the attacking host located in China, with a figure of 530 attacks and 23331 tuples. While China easily outperforms the other countries in terms of attacks and tuples in *UT*, the relative increase of the number of attacks and tuples not reported until after compensation from China is also relatively high. More specifically, the increase in the number of attacks from China is 29%, and for tuples the increase is a staggering 73%. China also dominates in the *CESNET* dataset, where 4444 attacks from China are detected only after compensation, and 309618 tuples. The respective gains are 29% and 39%. Second, for the *UT* dataset, we implicitly know the geographical location of the targets of attacks. Moreover, we know that traffic between hosts located in China and the UT campus network is often susceptible to packet loss. The same can be said for the United States, for which an 18% gain in terms of tuples can be observed. All these observations make us conclude that TCP control information and retransmissions are indeed strongly bound to the distance in geographical location between attacker and target, and that the effects on detection can be observed quantitatively.

Out of the top five attack origins in *UT*, the gain in the number of detected attacks from The Netherlands after compensation is at 119%. This gain is higher than the 29% for China, for example, while attackers in The Netherlands are located closer (from a geographical point-of-view) to UT's campus network. Investigation of the measurement data has shown that for attacks where the route between attacker and target is not impaired by apparent packet loss or high(er) latencies, the non-flatness of network traffic is caused mostly by packets that

Dataset	Logged attacks	TPR	FPR	TNR	FNR	*Acc*
UT	812	0.644	0.087	0.913	0.356	0.788
UT, compensated		0.784	0.096	0.904	0.216	0.849

Table 4.9: Detection performance in terms of attacks.

Dataset	Logged tuples	TPR	FPR	TNR	FNR	*Acc*
UT	4562	0.430	0.081	0.919	0.570	0.689
UT, compensated		0.585	0.090	0.910	0.415	0.758

Table 4.10: Detection performance in terms of tuples.

relate to TCP's *delayed ACK* mechanism and not by retransmissions, which is more likely for far-away countries.

Thus far we have shown the different detection results when using compensated and non-compensated data. However, we have yet to compare these detection results to our ground-truth, consisting of authentication logs from 58 hosts on the campus network of the UT. Since the ground-truth covers only a subset of the hosts considered before, the number of attacks and tuples reported in the remainder of this section is lower than reported in Table 4.7 and Table 4.8.

The detection performance of the detection algorithm in terms of attacks is shown in Table 4.9, where we again divide the results in both compensated and non-compensated. Analogously, the detection performance in terms of tuples in shown in Table 4.10. In both tables, we use the percentages of the previously introduced evaluation metrics. For example, the True Positive Rate (TPR) is the percentage of correctly identified attacks/tuples for which 5 or more sessions with authentication failures are reported in the ground-truth. The overall conclusion of the results is that compensation of the number of PPF yields a significantly improved TPR for both attacks and tuples. The TPR for attacks has improved from 64% to 78% for the *UT* dataset. For tuples, the figures are from 43% to 59%. These major improvements come at a minor cost in terms of false detections of roughly 1%. Also the accuracies for both attacks and tuples have improved significantly, from 79% to 85%, and 69% to 76%, respectively. We believe that the slight increase in the number of false detections is the case because of misclassified packets (e.g., unrecognized retransmissions), which in some cases cause benign network traffic to mimic dictionary attacks by becoming flat. These false positives are thus coupled to the performance trade-offs made in the plugin, as explained in Section 4.4.2.

4.5 Including SSH-specific Knowledge

In the previous sections, we have presented our approach for detecting the three phases of SSH brute-force attacks, and investigated under which circumstances this algorithm works on the Internet. To overcome the identified problems without changing our detection algorithm, we proposed a functional extension to flow export devices that exports information for compensating traffic that features network artifacts. Besides what has been presented before, we have learned from production deployments of our open-source IDS *SSHCure* that our detection approach has various shortcomings that ultimately cause compromises to remain undetected or false alarms to be raised. First, we have discovered characteristic features of the *OpenSSH* daemon that significantly impact the traffic between SSH clients and daemons. Second, attack mitigation systems on various network layers often yield traffic patterns that are similar to the definition of compromise traffic in the original algorithm.

In this section, we take a different approach to compromise detection, to address both identified shortcomings, among others. Instead of relying on the assumption that brute-force attack traffic is flat, we adapt our detection algorithm to the most-used SSH daemon: OpenSSH. By analyzing attack traffic that exhibits characteristics that are specific to OpenSSH, combined with an extensive analysis of attack tool behavior upon compromise, our detection algorithm is more resilient against unexpected deviations in network traffic.

4.5.1 Traffic Analysis

Our experience in analyzing SSH traffic has shown that network traffic between attacker and target can be affected at multiple stages: SSH daemon settings, attack tools and attack mitigation mechanisms. In the analysis presented in this section, we consider *OpenSSH* as the daemon running on attack targets, as it often comes preinstalled on Linux, BSD and Mac OS X operating systems. To verify whether it is the most used SSH daemon, we have performed a scan on the UT network, which has approximately 25k active hosts. Out of those, more than 700 hosts are running an publicly accessible daemon with a valid identification string, of which 97% identified itself as being *OpenSSH*.[13] Similar numbers on the OpenSSH market share are reported by other sources, such as 97% in [85] and 88% in [135].

4.5.1.1 OpenSSH daemon

The *OpenSSH* daemon features several configuration options that should be taken into account for the detection of SSH compromises:

[13] Other discovered SSH daemons were *SunSSH*, *Dropbear*, *Cisco SSH*, *Gene6*, *DesktopAuthority*, *SCS* and *WeOnlyDo*.

- `LoginGraceTime` defines the time after which the SSH daemon disconnects in case the client does not perform any more authentication attempts. This is done by sending a TCP `FIN` packet to the client. The default value is 2 minutes.
- `MaxAuthTries` defines the maximum number of authentication attempts per connection. The default value is 6. Note that many client tools, such as the OpenSSH client, close the connection already after 3 failed authentication attempts. The allowed number of authentications can be changed by the client as long as the value does not exceed `MaxAuthTries`.
- `MaxStartups` defines the maximum number of concurrent, unauthenticated connections. It is defined as the three-tuple `start:rate:full`, with default value 10:30:60. Rate-limiting in the form of dropping connections is then applied with a probability of `rate`/100 when more than `start` connections are unauthenticated. This probability increases linearly up to the moment in which `full` connections are unauthenticated.

Next to these settings, there are TCP settings that affect the network traffic between client and daemon. First, the value of the TCP `FIN`-timeout determines the maximum amount of time a TCP connection remains in the `FIN-WAIT-2` state. In this state, the daemon has initiated a connection termination and received a subsequent TCP `ACK` from the client. As soon as the client also closes the connection by sending a TCP `FIN` packet, the time between this packet and the previous TCP `ACK` packet determines the response of the server; If the TCP `FIN` packet is received before the TCP `FIN`-timeout has taken place, the server replies with a `FIN`+`ACK` packet, while a TCP `RST` packet is sent otherwise. This is shown in Figure 4.9. Second, the TCP keep-alive interval determines the maximum idle time of a TCP connection. In case of an idle connection, a TCP `ACK` packet is sent without payload. The OpenSSH daemon also has its own keep-alive mechanism, but since the TCP keep-alive mechanism is enabled by default, it is typically disabled.

Existence of users on the target system (i.e., the system where the *OpenSSH* daemon is running) does not affect the network traffic. From the attacker (client) side, no difference can be observed between an attempt using a valid username and an invalid password, and an attempt with an invalid username. Naturally, this is favorable in terms of security, as an attacker cannot determine whether a guessed username exists on the target host and thereby increase the probability of a successful authentication.

4.5.1.2 Attack Tools

We have analyzed ten uniquely identifiable tools for performing brute-force attacks, ranging from `expect`[14] scripts, to sophisticated applications that try to be

[14] `http://linuxcommand.org/man_pages/expect1.html`

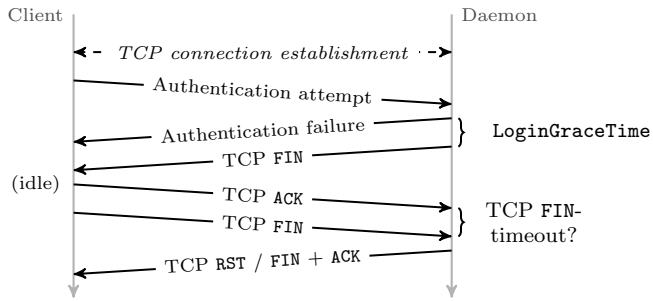


Figure 4.9: SSH client behavior after `LoginGraceTime`.

invisible for detection algorithms. These tools have been downloaded to our honeypots over several years, complemented by tools that we have found by searching the Web systematically; Five tools can be retrieved from their own Web page or Google Code, and four tools can be found on forums and other online communities.

Several observations have been made regarding the analyzed tools. First, the number of authentication attempts per connection varies between 1 and `MaxAuthTries`. Tools that perform a single attempt per connection are typically faster; Establishing one connection per attempt avoids waiting for the SSH daemon to report the authentication failure and the client to show a new prompt. Second, the number of PPF per login attempt very much depends on the configuration of the daemon, rather than the attack tool. Third, the attack tools' *action upon compromise*, i.e., the behavior of the attack tool with respect to the connection on which a successful authentication has taken place, was found to be the clearest indicator of a compromise, and is therefore key to our detection algorithm. This finding is crucial, as it makes the detection of compromises independent of the absolute number of PPF, which varies per attack tool and SSH daemon. We have identified four actions upon compromise:

- *Maintain connection, continue dictionary* – The connection with successful authentication is maintained, until the end of the attack. The attacker continues with the attack, also towards the compromised host.

- *Maintain connection, abort dictionary* – The connection with successful authentication is maintained, until the end of the attack. Other attack traffic towards the compromised host is stopped.

- *Instant logout, continue dictionary* – The connection with successful authentication is closed right after the compromise, while the attacker continues to attack both the compromised host and others.

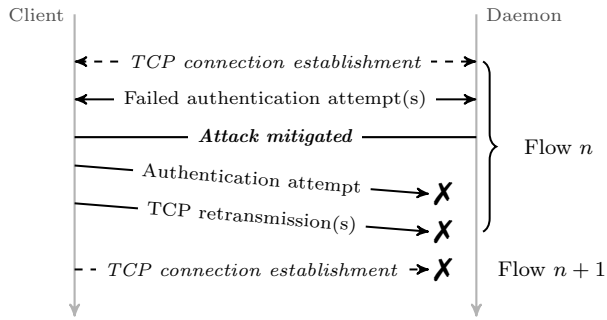


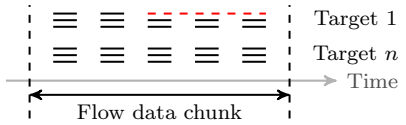
Figure 4.10: Attacker behavior after mitigation.

- *Instant logout, abort dictionary* – The connection with successful authentication is closed right after the compromise. Other attack traffic towards the compromised host is stopped.

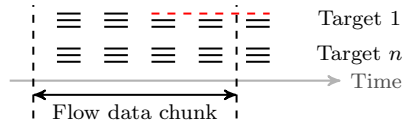
These actions form an integral part of the detection algorithm presented in Section 4.5.2. Actions featuring an instant logout are most prevalent in the considered tools.

4.5.1.3 Attack Mitigation Algorithms

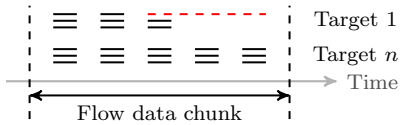
Several SSH attack mitigation mechanisms have been developed over the years, to reduce the risk of compromises during a brute-force attack. These mechanisms exist for both the host-level and the network-level. Host-level mechanisms, usually software-based, scan authentication log files and as soon as the number of failed authentication attempts exceeds a threshold, traffic from the attacker is blocked. Blocking can be performed on several layers. First, on L5, tools like *denyhosts* still allow TCP connections to be established to the target device, while setting up SSH connections from the attacking host is prohibited. Since no packets are dropped at the connection-level, no retransmissions or failing connection establishments can be observed. Second, tools like *fail2ban*, *sshdfilter* and *SSHblock* operate on L4 by instructing a local firewall to block traffic from the attacker to the target. If mitigation takes place while a TCP connection is active, retransmissions will occur. Also new TCP connections to the target cannot be established anymore, resulting in `SYN`-only flows. Both situations are shown in Figure 4.10, where the number of PPF of Flow n deviates from typical brute-force flows, due to the additional packets involved in the retransmission(s). After *Flow* n, there will be at least one `SYN`-only flow (Flow $n + 1$). Third and last, tools like *SSHGuard* drop any traffic from the attacker's IP address using a local firewall, i.e., at L3. From a network traffic perspective, the behavior is identical to a L4-block.



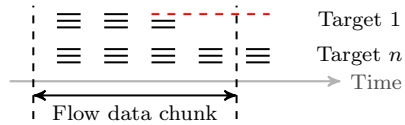
(a) Maintain connection, continue dictionary (1)



(b) Maintain connection, continue dictionary (2)

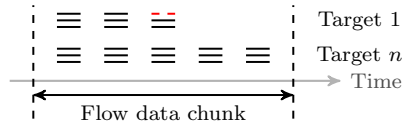


(c) Maintain connection, abort dictionary (1)



(d) Maintain connection, abort dictionary (2)

(e) Instant logout, continue dictionary



(f) Instant logout, abort dictionary

Figure 4.11: Various types of compromise flows in a chunk of flow data.

Besides host-level mitigation mechanisms, also network-level mechanisms can be in place. These mechanisms are usually operated by packet forwarding devices, performing some sort of traffic blocking, e.g., by means of Access Control Lists (ACLs) or null-routing. Blocking rules can be composed based on blacklists or detections on honeypots, for example. The network traffic after mitigation is similar to host-level mitigation on L3 or L4.

4.5.2 Algorithm

Key to our compromise detection are the four actions that can be observed after a compromise. We have transformed these actions into six scenarios, as shown in Figure 4.11. The two additional scenarios have been defined to accommodate for the fact that many analysis applications receive and process flow data in fixed-size time bins, as a consequence of which our algorithm has to take into account that attack data may be spread over multiple data chunks. Each of the subfigures shows a flow data chunk, with flows (long dashes) towards targets running an SSH daemon. Short-dashed lines mark a flow with a compromise.

In Figure 4.11a, we show that the compromise flow is maintained until the end of the attack, and that other login attempts are observed in parallel towards the same target. A similar scenario is shown in Figure 4.11b, but since the end

of the attack does not lie within the current data chunk, the compromise flow is characterized by an unterminated TCP connection (i.e., without a TCP `FIN` or `RST` flag set). Similarly to these two scenarios, we show in Figure 4.11c and 4.11d how the compromise flows should be identified in case the attacker aborts its dictionary towards the compromised target: traffic from the same attacker towards other targets reveals the end of the attack. Figure 4.11e and 4.11f show situations where the attack tool performs an instant logout upon compromise. Observe that the compromise in Figure 4.11f may also be very close to the end of the data chunk, which is why compromises classified according to this scenario are checked in the next data chunk again, to verify whether there is no traffic from the attacker towards the compromised target.

The compromise detection can be summarized in two steps:

- **Step 1 – Matching traffic against scenarios**. As soon as a *brute-force* phase has been detected between attacker and target, this phase aims at detecting one of the scenarios shown in Figure 4.11. In case of a match, a compromise is detected. Special care must be taken with unterminated connections, as shown in Figure 4.11b and 4.11d, as they may be the result of an attacker stopping its attack without properly closing all connections, instead of a compromise. If this is the case, the SSH daemon will timeout and close the connection after `LoginGraceTime`, as discussed in Section 4.5.1.1. At the flow-level, this can be verified by checking the duration of the return flow, i.e., the flow from target to attacker. In case its duration matches the configured `LoginGraceTime`, the attack was stopped without a compromise. Otherwise, we consider the target host to be compromised.
- **Step 2 – Identification of mitigation mechanisms**. After identification of a matching scenario, the traffic is checked for signs of activated mitigation mechanisms. As soon as these mechanisms are activated, at least one of the following situations may apply: 1) Mid-connection mitigation can result in a number of PPF that is higher than the identified *baseline*, due to retransmissions between attacker and target, or 2) new connections have merely a TCP `SYN` flag set and typically consist of three packets, which is a frequently used retry count for establishing TCP connections. Identification of mitigation mechanisms is crucial, as they would trigger false positive compromise detections (i.e., those that feature an instant logout) otherwise.

4.5.3 Validation

In this section, we present the validation of our detection algorithm. We start by describing our two datasets in Section 4.5.4. To be able to evaluate the algorithm, we have implemented it as part of *SSHCure* v2.4. The validation results are discussed in Section 4.5.5. After that, we evaluate the performance of *SSHCure* as a system in Section 4.5.6.

4.5.4 Datasets

We collected two datasets on the campus network of the UT. The network features a publicly routable Class B IPv4 address block, of which typically 25k addresses are actively being used. Both datasets comprise a period of one month, collected in November and December 2013, and January and February 2014, respectively, consisting of the following data:

- *Honeypot log files* – These have been collected from various low-, medium- and high-interaction honeypots.
- *Workstation & server log files* – These have been collected from workstations and servers, which all have a publicly accessible SSH daemon.
- *Flow data* – Flow data with a sampling rate of 1:1 has been collected at edge routers and, as such, contains all SSH traffic entering and leaving our campus network.

The exact composition of the datasets is shown in Table 4.11, which shows the number of honeypots, servers and workstations of which the log files comprise the dataset, and the number of attacks identified.

The datasets have been chosen carefully to reflect two completely different types of systems. On the one hand, dataset D1 solely consists of data from honeypots, i.e., systems set up for being compromised (easily). On the other hand, dataset D2 is made up of data from mostly servers, which are likely configured with very strong passwords and attack mitigation mechanisms. In addition, we deliberately selected workstations, servers and honeypots that are operated by different persons, to ensure that our results are not biased by similar configurations.

Since the collected log files are the one and only proof of whether login attempts have succeeded or not, we consider this data the ground-truth for the validation of this work. More precisely, we consider all successful authentications after more than six login attempts and have no idle period of more than one hour to be compromises, and exclude all logins from hosts in the IP address range of the UT. Note that in a benign situation, six attempts will typically be carried by two flows (due to `NumberOfPasswordPrompts` being set to 3 in the configuration of the OpenSSH SSH client), while up to six flows may be observed during an

Dataset	Honeypots	Servers	Workstations	Attacks
D1	13	0	0	632
D2	0	76	4	10,716

Table 4.11: Dataset composition in terms of device types and attacks.

attack, in case an attacker performs only one authentication attempt per connection. Since our goal is to validate the detection of compromises, we consider only those attacks that show *brute-force* phase behavior.

4.5.5 Detection Accuracy

We have compared the log files to *SSHCure*'s detection results based on the following metrics:

- True Positives (TPs), False Positives (FPs): Attacks correctly and incorrectly identified to feature a compromise, respectively.
- True Negatives (TNs), False Negatives (FNs): Attacks correctly and incorrectly identified to not feature a compromise, respectively.
- *Accuracy* (*Acc*): This metric has already been defined in (4.1).

The results of our evaluation are shown in Table 4.12, where we list the values for all evaluation metrics, as well as their respective rates/percentages. For example, the *True Positive Rate* (TPR) is the percentage of attacks correctly identified to feature a compromise.

Although the number of correctly classified attacks in D1 is very high, yielding an accuracy of 84%, we have to face incorrect classifications as well. On the one hand, FPs are mainly a result of the sensitivity of the *instant logout, continue attack* scenario (Figure 4.11e). In this scenario, we observe deviations in the number of PPF from the identified *baseline*. We have found to need a slightly higher sensitivity to obtain the results in Table 4.12, than for servers, for example. The sensitivity may be reduced for honeypots, which will reduce the FPs, while the number of FNs increases. On the other hand, the FNs are caused by the result of the nature of honeypots, and show how the characteristics of the dataset limit our approach; The easy-to-guess credentials of honeypots can result in compromises from the first authentication attempt or even compromises after every attempt. As such, the *baseline* for identifying deviations in the number of PPF, i.e., compromises, cannot be established reliably.

The explanation for incorrectly classified attacks in D1 is confirmed by the evaluation results of D2, where no compromises are captured. We assume that this is due to the low number of workstations compared to servers considered, as server administrators typically have more system administration skills than

Dataset	**TP**	**TN**	**FP**	**FN**	***Acc***
D1	157 (0.692)	374 (0.921)	32 (0.079)	70 (0.308)	0.839
D2	0 (–)	10327 (0.997)	26 (0.003)	0 (–)	0.997

Table 4.12: Validation results per dataset.

the average workstation user, typically resulting in stronger login credentials. However, a much higher accuracy (close to 100%) is achieved for D2, due to the fact that only 0.3% of the attacks is incorrectly classified.

4.5.6 SSHCure Performance

We are aware of many successful deployments of *SSHCure*, in networks at different scales: campus networks, hosting companies, Internet Service Providers (ISPs), and CSIRTs up to government-level. We also closely collaborate with CESNET, which uses *SSHCure* on a central flow collector where flow data from all peering links is collected. Given that we aim at deploying *SSHCure* in high-speed networks, we have evaluated whether *SSHCure* is able to analyze CESNET's SSH traffic in real-time, i.e., every data chunk should be processed before the next data chunk arrives. In January 2014, where up to 29.9 GB of SSH traffic per five minute data chunk has been transferred, *SSHCure* was able to do so for the vast majority of data chunks. Only in situations with many large and concurrent attacks, *SSHCure* was not able to finish in time on our measurement system, after which it automatically skipped the next data chunk to not overload the collection system.

In addition to evaluating *SSHCure*'s processing performance, we have compared its detection results to the OpenBL SSH blacklist. Since comparing IDS detection results to a public blacklist may be misleading [159], these numbers are however merely indicative. OpenBL deploys more than 40 sensors that report which host has performed brute-force login attempts on port 22. With only a single instance of *SSHCure* deployed at the UT, we already achieve a coverage of OpenBL of up to 3% per day over January 2014, defined as the share of IP addresses blacklisted by OpenBL that was also reported by *SSHCure*. By deploying *SSHCure* in the CESNET network, we even achieve a coverage of up to 7% per day with a single sensor. In addition, depending on the selected day of the month, 14-37% of the attacker IP addresses reported by *SSHCure* at UT was not (yet) blacklisted by OpenBL, while in the CESNET network, this percentage was 47-95%. We therefore envision *SSHCure* to be used as a complementary sensor for SSH blacklisting.

4.6 Conclusions

In this chapter we have shown that compromise detection for SSH is viable for deployment on the Internet. To prove this, we started in Section 4.3 with an elementary approach that relies on the observation that brute-force attacks typically feature up to three phases. This approach is based on preliminary work and similar to brute-force attack detection approaches proposed by others. Its novelty lies in the detection of the *compromise* phase, which was not touched by other works in the extent in which it is covered in this thesis. Our elementary

approach is based on theoretical assumptions of brute-force attacks and it has been tuned based on measurements in lab environments. However, we found that improvements are necessary for it to be usable in practice, i.e., on the Internet.

As a follow-up of our elementary approach, we added two crucial and new components. First, in Section 4.4, we investigated how network artifacts impair the detection of the *brute-force* phase. These network artifacts, caused by TCP retransmissions and control information, were found to be the source of these problems. To overcome the impairments of the reported TCP phenomena, we have extended the set of fields exported in flow data, such that the phenomena can be discriminated. Without any change to the algorithm used for our elementary approach, our detection accuracies increase from 79% to 85%, and 69% to 76%. This leads us to conclude that many of the possible flow monitoring applications mentioned in [98] can benefit from this work. From talks with a vendor of flow export devices we have even learned that adding a selection of the statistics used in this work to flow data is of interest to many customers, and that efforts are being undertaken to do so in their products. Second, in Section 4.5, we included domain-specific knowledge on SSH to make our detection algorithms capable of recognizing SSH protocol and tool behavior that would otherwise be classified erroneously as a compromise.

To validate whether our compromise detection approach for SSH also works in practice, we developed the presented algorithms as part of our open-source IDS *SSHCure*. Based on large-scale validations on the campus network of the UT using *SSHCure*, we conclude that detection accuracies close to 100% can be achieved, once we make our compromise detection specific to the SSH protocol. Additionally, we distributed *SSHCure* to a large number of companies, institutes and ISPs, such that they could share their operational experience with our compromise detection algorithms with us. Both our own experience and experiences of others make us conclude that flow-based compromise detection for SSH is viable and works well in practice and production. This is underlined by the fact that *SSHCure* is used in a considerable number of production deployments all over the world.

We are aware of more stealthy attacks that feature a very low intensity that might stay under the radar of IDSs, such as those algorithms presented in this thesis. Work on detecting such attacks has been done before, e.g., in [18], [51], but these works are typically not flow-based. We therefore consider the detection of low-intensity attacks as an important direction for future work.

Compromise Detection for Web Applications

In the previous chapter, we have shown that flow-based compromise detection can successfully be applied for SSH. We therefore investigate in this chapter whether flow-based compromise detection can be performed for other protocols as well. Given that Web applications in general and CMSs in particular are popular targets of brute-force attacks, we target these in this chapter. Similar to what we presented in Chapter 4, we started with an elementary approach that aims at identifying attack phases based on attack signatures. The main lesson learned of the work describing this approach ([71]) was that signatures of attacks against Web applications may be used for detection, but more research is needed to reduce the number of false detections as a consequence of traffic that consists of many small connections, such as traffic generated by Web crawlers, calendar fetchers, and photo galleries. In this chapter, we use an approach based on per-connection histograms that provide information on packet payload sizes in flow data, on which we use clustering methods for grouping similar connections. We intuitively believe that this allows us to differentiate between traffic classes, so between attack traffic and benign traffic. Additionally, the use of histograms should provide a means to recognize the network artifacts investigated in Section 4.4 and overcome any consequent deficiencies. The main contribution of this chapter are Web application compromise detection algorithms, which we validated using prototype implementations that we deployed in the production network of a large Dutch Web hosting company.

The papers related to this chapter are [50], [71]. Since [71] is an early pioneering work on the subject presented in this chapter and provides only elementary results, we present mainly the work in [50] and reference [71] only where applicable.

The organization of this chapter is as follows:

- *Section 5.1 presents the background on the work presented in this chapter, as well as this chapter's contributions.*
- *Section 5.2 introduces the concept of using histograms for intrusion detection. These histograms provide more details about individual packets in flows and help us to overcome several deficiencies of regular flow data, such as not being able to discriminate retransmissions.*

- *Section 5.3 describes how we use histograms and clustering techniques for our brute-force and compromise phase detection algorithms.*
- *Section 5.4 describes our validation approach and datasets, and discusses our validation results.*
- *Section 5.5 concludes this chapter.*

5.1 Background

Web applications have become one of the most popular targets of brute-force attacks in recent years. A prime example of a class of Web applications that gained lots of attention by both users and attackers are CMSs, such as Wordpress, Joomla, and Drupal. They provide a means to build Web sites to anyone, even people with little technical knowledge. The popularity of CMSs, and these in particular, is also underlined by numbers: In 2015, almost 30% of all Web sites on the Internet are built using these CMSs.[1] The widespread use of these CMSs also comes with a risk: The fact that anybody can use them, even people with limited technical skills that are unaware of security threats and measures, leads to outdated and vulnerable CMSs, and the use of weak administrator passwords [34]. As such, CMSs end up being a prime attack target and the number of attacks is increasing every day, as shown and explained in Chapter 1.

Detecting attacks against Web applications can be done in several ways. From talks with several Dutch Top-10 Web hosting companies we have learned that the detection of attacks in a host-based fashion is by far the most popular approach. This can be done, for example, using authentication monitors that analyze log files on Web servers on-the-fly and block attackers by IP address after a certain number of failed authentication attempts. These monitors come with so-called Web panels – administration interfaces for Web hosting products. The fact that all three major Web panels, namely cPanel, Plesk and DirectAdmin [112], provide this functionality, indicates that this specific form of attack detection is used by many Web hosting companies. Another example of a host-based approach is the protection of attacks on the level of Web applications, such as CAPTCHA and IP-based authentication blockers, typically to be installed and configured by the Web application user instead of the Web hosting company. These host-based approaches, where monitoring is run on the same infrastructure as the monitored services, are however vulnerable to attacks, as attacks may lead to a significant increase in load on the infrastructure.

[1] `http://w3techs.com/technologies/overview/content_management/all`

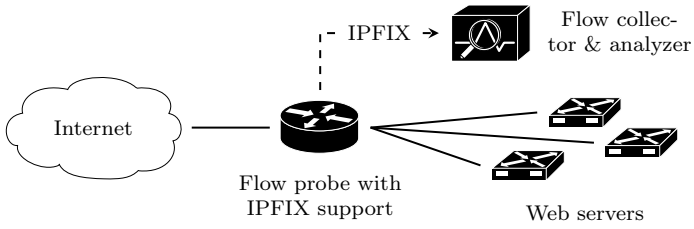
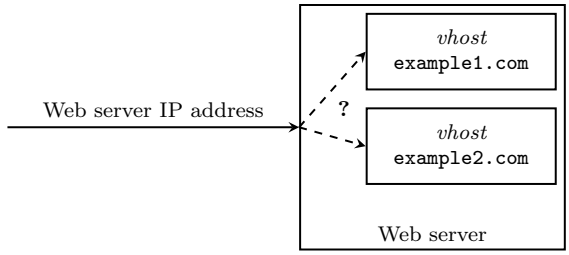
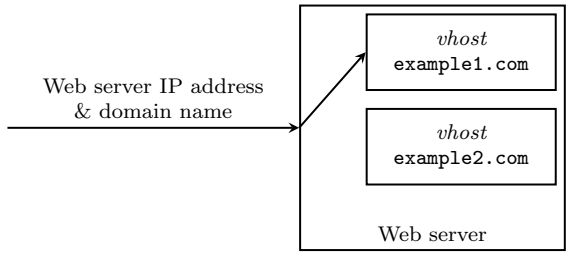


Figure 5.1: Flow monitoring for Web servers using IPFIX.



(a) Scan using Web server IP address



(b) Scan using Web server IP address and domain name

Figure 5.2: Multiple *vhost*s on a Web server.

To overcome the problems of host-based detection, we take a network-based approach for which only a single sensor needs to be deployed at a strategic observation point. A network-based approach can be implemented in two ways; We either take a specific intrusion detection system, such as Snort or Bro, or we take a more generic approach that is based on established monitoring technologies/protocols, such as IPFIX, for which the collected data can also be used for other applications than intrusion detection. We take the more generic approach based on flow data, where individual packets are aggregated into flows by a flow exporter/probe, and sent to a flow collector for storage and analysis, as shown in Figure 5.1 and explained in Chapter 2.

In this chapter, we use our three-phase attack model again, describing the *scan*, *brute-force* and *compromise* phases of a brute-force attack. The *scan* phase in the context of attacks against Web applications is different in nature from attacks against other applications. This is because Web applications can typically not be scanned using target IP addresses only, as domain names are required to reach Web applications. This is illustrated in Figure 5.2. Web applications are served by *vhost*s, which can be considered as virtual containers on a Web server, one per domain, such that one Web server can serve multiple domains. Mapping IP addresses to *vhost*s is non-trivial and not directly related to the attack itself, which is why we ignore the *scan* phase in the remainder of this work.

5.1.1 Related Work

In the previous chapter, we have shown how to detect SSH compromises using flow data. In contrast to SSH, the Hypertext Transfer Protocol (HTTP) protocol provides many more combinations of actions to be performed by an attacker. This can best be explained by an example. The SSH protocol requires successful authentication to be able to perform any further action, i.e., to advance in the protocol's state machine. On the contrary, the HTTP protocol does not have any such mechanism and (form-based) authentication is handled as any other payload to the protocol. The consequence is that flat traffic does not need to be present in large badges anymore, as attackers may perform other actions, such as triggering a page load or redirect, to evade detection.

Several works, such as [80], have targeted the anomaly-based detection of attacks over HTTP(S), but always rely on individual packets for doing so. The only flow-based attempt in this direction is described in [71], where the authors extract attack signatures from tools that can be used for performing brute-force attacks against Web applications. The major disadvantage of this signature-based approach is that it has shown to catch legitimate traffic as well, mostly caused by calendar fetchers and Web crawlers. This is because the approach is based on the assumption that brute-force attacks consist of many flows that are rather small in terms of packets and bytes, which is however also true for the aforementioned type of Web applications. Although our approach also assumes attack traffic in the *brute-force* phase to be flat, we design our detection approach such that attack traffic can be discriminated from benign traffic that is alike in terms of packets, bytes and durations, by means of using histograms for intrusion detection.

5.1.2 Challenges & Contributions

Existing works have shown that promising detection results can be achieved by analyzing flow data for brute-force attacks [37], [74], [75], [79], [83]. However, as shown in Section 4.4, fluctuations in network traffic, such as TCP retransmissions and control information, may result in both false positives and false negatives. To overcome these problems, the novelty presented in this chapter is to analyze flow data that is enhanced with histograms that describe packet payload distributions. These histograms show not only the total size of a specific flow, as is the case for regular flow data, but the entire payload size distribution. For this purpose, we use a flow exporter, or flow probe, which we have equipped with an extension for exporting histograms for every observed flow, to aid in our detection of attacks against Web applications.

The contributions of this chapter can be summarized as follows:

1. We present a network-based approach for detecting brute-force attacks and compromises against Web applications, based on IPFIX, that is resilient against attacks on the core hosting infrastructure. (Section 5.3)

2. The use of histograms for detection makes our approach resilient against real-world artifacts introduced by TCP, such as retransmissions and control information, even if the network traffic is encrypted. (Section 5.2)

3. This work has been validated using a month-long dataset of a Dutch Top-10 Web hosting company. The dataset consists of flow data and Web server log files for approximately 2500 Web hosting accounts. (Section 5.4)

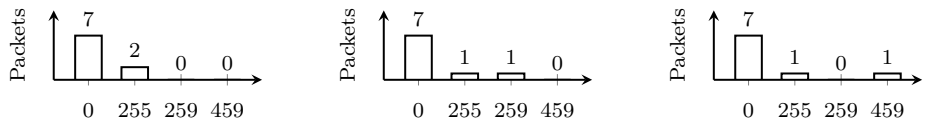
5.2 Histograms for Intrusion Detection

In this section we demonstrate how histograms can be used for intrusion detection. We start by covering background information on histograms and motivating their use in Section 5.2.1. Then, in Section 5.2.2, we explain how to compare histograms and form clusters, while in Section 5.2.3, we provide several concrete examples and measurement-based insights into how indicative attack traffic is when represented using histograms.

5.2.1 Traffic Characteristics

Key to identifying brute-force attacks in flow data is to aggregate similar records into clusters. Ultimately, records describing the same attack, which are assumed to be rather similar in terms of the number of packets and bytes, and duration, should be part of the same cluster. Several works have however shown that relying on packet and byte counters in flow data should be done with care, especially when it comes to identifying *flat* traffic for network security analysis. For example, it has been shown that TCP retransmissions and control information, which affect the packet and byte counters due to timing parameters, cannot be discriminated in flow data [52]. This causes attacks from countries that are far-away from the observation point – above all in terms of geographical distance – to stay under the radar of IDSs. But even if flat traffic is identified properly, its "detection for HTTP(S) was found to be ineffective, because valid AJAX updates common on Web 2.0 tend to produce flat traffic pattern" [37]. This is also confirmed by [71], where it is shown impossible to differentiate traffic of Web crawlers and calendar fetchers from dictionary attack traffic based on packet and byte counters alone. We therefore reiterate the call in [52] that flow data must be enhanced by additional fields to become a reliable source of information for intrusion detection.

To overcome the described problems with counters in flow data, we need more granular information on individual packets in a flow. Several metrics could be used for this purpose. For example, we have experimented with exporting packet inter-arrival times in histograms, as it was shown that timing information can be used for identifying applications in flow data [25]; Similar application-layer actions, such as login attempts, would feature similar timing characteristics, allowing for the aggregation of attack flows into the same cluster. We have however found two problems with this approach. First, inter-arrival times do not allow



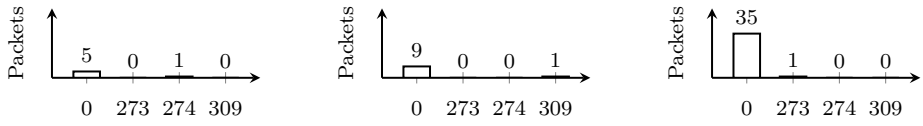
(a) Histogram A (b) Histogram B (c) Histogram C

Figure 5.3: Payload size histograms in typical HTTP brute-force attack traffic.

TCP control information packets to be discriminated from other packets, yielding this approach rather susceptible again to the problems described in [52]. Second, reliable measurements can only be done using special clocks for accurate hardware timestamping, which are neither available in our measurement infrastructure, nor in many others. We have therefore found this approach to be too sensitive to measurement errors, impairing subsequent clustering procedures.

Along with the other previously explained problems, the problems of inter-arrival times can be overcome by using per-flow information on individual packet payload sizes. This even allows us to discriminate TCP control information in flow data, as it is often carried in zero-payload packets. Also, we know that Web crawlers, calendar fetchers and dictionary attacks, for example, use different distributions of packets within flows, which allows us to discriminate these applications using the more granular data. Furthermore, since encrypted channel handshakes result in a constant pattern in every connection, the use of histograms allows for clustering similar encrypted channels. For these reasons, we export – per flow – a histogram with packet payload sizes, such that each *sample* in the histogram represents the payload size of one packet in that flow. To do so, we defined an enterprise-specific IE for IPFIX and instrumented our measurement infrastructure with it. Based on the datasets described in Section 5.4.2, we can conclude that our payload histograms extend the size of a flow record (on the wire) by at most 37 bytes in 99% of all flow records.

The use of histograms for intrusion detection in general is not new. An extensive overview is given in [14], where it is explained how to map network traffic features to histograms, cluster these histograms, and classify anomalous traffic patterns based on the created clusters. While the authors provide examples of various common network traffic anomalies, such as port scans, they demonstrate the generic viability of their histogram-based approach, but do not focus on a specific application or extension as we do in this chapter for flow-based brute-force attack and compromise detection. Also, we use a pivotal distance metric for clustering that is not covered in [14], for reasons to be discussed in the next subsection.



(a) Histogram D (b) Histogram E (c) Histogram F

Figure 5.4: Payload size histograms in benign HTTP flows.

5.2.2 Clustering Histograms

Clustering aims at grouping objects with similar characteristics into sets, such that the sets feature a low inter-set similarity (i.e., large distance between objects in different sets) and a high intra-set similarity (i.e., small distance between objects in the same set). When histograms are used for intrusion detection, it is key to cluster histograms that are 'similar' with respect to their bins. Alternative to maximizing the similarity of histograms within a cluster is to minimize the distance between histograms. Well-known and frequently-used distance metrics are the *Manhattan* (L1-norm), *Euclidean* (L2-norm) and *Mahalanobis* distances [14], [62]. The main problem with these distance metrics in the context of our work is that they fulfill the *shuffling invariance* property, meaning that the distance between any two histograms does not change when bin values are interchanged. This problem can be observed in Figure 5.3, which shows three payload size histograms in a typical HTTP brute-force attack. Intuitively, we would assume the distance between Histograms A and B to be smaller than the distance between Histograms A and C, given that the difference between 255 and 259 is much smaller than the difference between 255 and 459. This is however not the case for distance metrics that satisfy the *shuffling invariance* property, as is demonstrated for the Euclidean distance:

$$D(x,y) = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2} \tag{5.1}$$

Using (5.1), we can calculate the distances between histograms in Figure 5.3:

$$D(A,B) = \sqrt{|7-7|^2 + |2-1|^2 + |0-1|^2} \tag{5.2}$$

$$D(A,B) = D(A,C) = \sqrt{2} \tag{5.3}$$

Note that both distances are equal, even though the histograms differ significantly, especially when they are visualized to scale. Once we translate this result to network traffic, the unsuitability of the (Euclidean) distance metric becomes clear immediately; The difference between Histogram A and B can easily be caused by variability in the TCP header or differences in username and password lengths, for example, while Histogram C shows significantly different traffic.

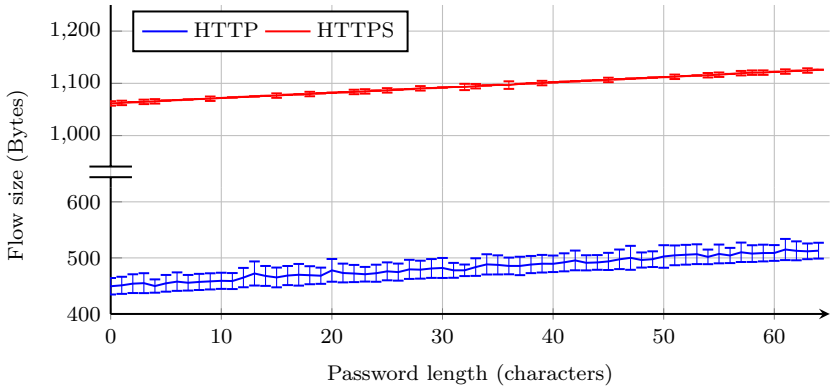


Figure 5.5: Impact of password length on total flow size.

A solution to this 'problem' is provided in [5], where the Minimum Difference of Pair Assignments (MDPA) distance metric is defined. In a nutshell, MDPA aims at *finding the minimum difference of pair assignments between two sets*, where *sets* are histogram bins in our context:

$$D(x,y) \quad = \quad \min_{x,y}\left(\sum_{i,j=0}^{n-1} d(x_i,y_j)\right) \tag{5.4}$$

Here, $d(x_i, y_j)$ is defined as the arithmetic difference between bin i in histogram X and bin j in histogram Y. Hence, the more similar any two histograms are, the smaller the value D. For the histograms in Figure 5.3, the following distances can be obtained: $D(A,B) = 4$ and $D(A,C) = 204$, as demonstrated in Appendix A. From these results it becomes clear what the added value of the MDPA distance metric is, compared to commonly-used metrics like Euclidean. The MDPA distance metric does not satisfy the *shuffling invariance* property, but, besides, is similar in nature to the commonly used distance metrics. We therefore rely on this metric in the remainder of this work, unless indicated differently. For a detailed MDPA example, we refer the reader to Appendix A.

5.2.3 Measurements

Based on the concept of using histograms for intrusion detection and calculating inter-distances for clustering similar histograms, we provide in this section several measurement-based insights into how this works in practice.

In Figure 5.4, we visualize the payload size histograms of three consecutive flows in a benign client-server interaction with a Web shop. What catches attention are the zero-value bins, which appear to be relatively large in some histograms, such as for Histogram F. Packets accounted in the zero-value bin feature

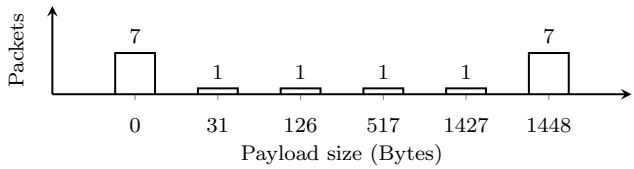


Figure 5.6: Payload size histogram of a typical TLS flow.

no payload, meaning that they are likely made up of TCP acknowledgements, window updates and other control information. What all histograms in Figure 5.3 and 5.4 have in common are the seemingly minor deviations in the number of bytes. To investigate whether – and if so, until which extent – the password length of authentication attempts affects the size of a flow, we have measured various password lengths and resulting flow sizes in a lab setup. We have done this by generating – per selected password length – 100 random passwords that were fed into the *Patator*[2] brute-force attack tool, and captured the resulting traffic. The results, which are shown in Figure 5.5 together with their respective standard deviations, indicate that every password character accounts for one byte in the total flow size. Deviations in flow sizes are found to be mainly the result of TCP sometimes dividing the returned Web page over two segments instead of one, resulting in additional acknowledgements. Given that most popular passwords, which are naturally also commonly found in *dictionaries*, typically feature no more than ten characters [110], we conclude that the impact of password lengths on the total flow size is limited.

Another reason for histograms to appear significantly different is when they represent packet payloads in Hypertext Transfer Protocol Secure (HTTPS) connections, as opposed to histograms for regular HTTP connections. To investigate how different the resulting histograms are, we have also measured attacks over HTTPS in a lab setup. As shown in Figure 5.6, HTTPS connections are bin-wise fundamentally different from their non-encrypted counterparts when compared to the histograms in Figure 5.3 and 5.4: Both the total number of bins and the total flow size are larger, while the number of zero-payload packets is basically the same. The differences can be accounted to the establishment of the encrypted channel, which requires cipher selection, key exchange, etc. However, when considering histograms of benign client-server interactions and brute-force attacks in HTTPS, a characteristic difference in distances similar to that in the case of HTTP traffic can be observed.

The most important take-away from the measurements and histograms presented in this section is that histograms for benign connections are quite different from histograms in typical brute-force attacks, such as those visualized in Figure 5.3. We will use this observation in our detection approach, which we explain

[2]Patator v0.7, which can be retrieved from `https://github.com/lanjelot/patator`

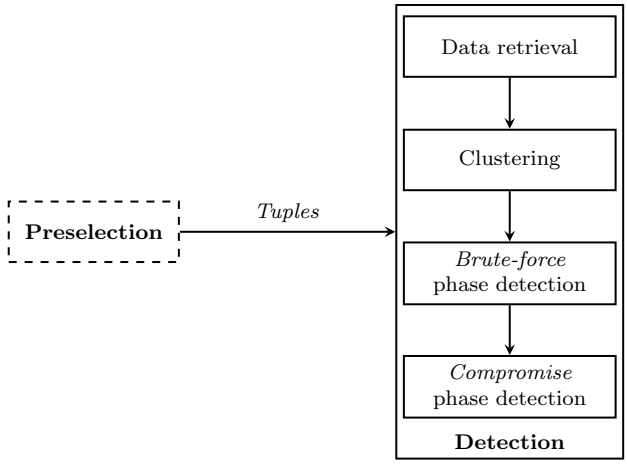


Figure 5.7: Conceptual detection approach.

in the next section, to discriminate *brute-force* phase traffic from other traffic and identify any subsequent compromises.

5.3 Detection Approach

Our detection approach is based on data exported by a flow exporter using IPFIX, and consists of two phases: *Preselection* and *Detection*. The *Detection* phase in itself is made up of several steps, as is depicted in Figure 5.7. Both phases, and the respective steps they comprise, are explained in the remainder of this section.

As with most flow-based analysis applications, the various analysis steps operate on flow data chunks. Data chunks consist of flow data that has been received in fixed-length time intervals, typically in the order of several minutes. The use of data chunks allows for near-real-time processing of network traffic on the one hand, and demarcates the dataset used per iteration on the other.

5.3.1 Preselection

The *Preselection* phase serves to make a rough data selection based on a number of criteria, such that the amount of data to be processed in further steps is reduced. Since this has advantages solely in terms of performance, this is an optional step. If used, *Preselection* returns a list of IPv4 and IPv6 address tuples of possible attackers and targets. More formally, we define a *tuple* as a pair of

source and destination IP addresses, source port number and *vhost*.[3] To qualify for preselection, an attacker must have generated at least N flows towards a target. Note that we refer several times to this number in the remainder of this chapter and that the used value for N is explained in Section 5.4.

Since *Preselection* targets data filtering and not detection, exceeding the threshold for tuple qualification is often easy, even for benign applications, such as Web crawlers and calendar fetchers. It is then up to the *Detection* phase to classify these cases as benign. Nevertheless, our measurements have shown that *Preselection* can improve the detection process' overall performance by more than a factor 7 in terms of processing time on our validation datasets. This is because the clustering procedure, covered in Section 5.3.3, is by far our most computationally expensive component, so its use should be limited as much as possible by confining the input dataset.

5.3.2 Data Retrieval

This step retrieves the flow data used within the *Detection* phase. In case a *Preselection* was done before, only data that belongs to the preselected tuples within the current data chunk is retrieved, and a full data chunk otherwise. After retrieval, flow data that cannot be part of an attack by definition is filtered out. For example, flow records from attacker to target typically feature at least four packets, because every valid HTTP request consists of the following packets at least:

- TCP `SYN` – First packet of three-way handshake.
- TCP `ACK` – Third packet of three-way handshake.
- HTTP `GET/POST` – Actual HTTP request.
- TCP `FIN/RST` – Connection teardown.

Note that network flows are typically unidirectional in nature, so these packets represent merely the traffic from attacker to target. Since we can only identify TCP flags in flow data and not whether a flow actually features an HTTP request, we use the TCP `PSH` flag. This flag signals an application-layer data exchange, so we consider a new connection to a common Web server port (80, 443) to feature an HTTP request. We filter out every flow that does not feature at least TCP `SYN`, `ACK`, `PSH` and `FIN/RST` flags. The retrieved data is presented to the next step, *clustering*, per tuple.

[3]Many IPFIX flow exporters extract *vhosts*, often referred to as *HTTP hostname*, from HTTP headers. This information is no prerequisite for our detection approach and therefore only used within the *Preselection* phase.

5.3.3 Clustering

Histograms of attack traffic are very much alike when attacks reside in the *brute-force* phase, since repeated application-layer behavior results in flat traffic, as shown in Section 5.2. As a consequence, this will cause *brute-force* traffic to be clustered. For the *compromise* phase, however, we analyze precisely the traffic that falls outside the cluster featuring *brute-force* phase histograms. We avoid any clustering impairment caused by TCP protocol variability (e.g., control information segments), as described in [52], by removing the zero-value bins from histograms; Potential traffic variability is typically caught in zero-value bins as related segments do not feature any payload.

In this work, we use Hierarchical Cluster Analysis (HCA), which aims at building clusters based on inter-cluster distances in a hierarchical fashion [84]. HCA uses either one of the following strategies: *divisive* (also commonly referred to as *top-down*), or *agglomerative* (*bottom-up*). We take the agglomerative approach, because it is faster than divisive clustering for larger datasets if the entire hierarchy needs to be built. Since histograms describing compromises are assumed to be outliers compared to histograms describing brute-force traffic, we would need to 'divide' all the way down to individual histograms to find potential 'compromise outliers' in case of divisive clustering.

The advantage of using HCA is that, unlike other clustering approaches such as k-means, HCA does not require the number of clusters to be set in advance. Instead, HCA can stop the clustering process (referred to as *stop linking clusters* in HCA jargon) as soon as certain criteria are no longer satisfied. An example criterion is that the distance between the selected pair of observations for cluster linkage is above a given threshold. Linking clusters is done based on a linkage method and a distance metric. The linkage method determines which two histograms from two potentially to be linked clusters to apply the distance metric to (i.e., which histograms to use for inter-cluster distance calculation), as each of the two clusters potentially contains multiple histograms. In this work, we rely on *single-linkage* as the linkage method, and on MDPA as the distance metric, as discussed in Section 5.2. Single-linkage selects the two least dissimilar histograms in two clusters to determine the inter-cluster distance. The choice for single-linkage, as opposed to *complete-linkage* (which considers the two most dissimilar histograms), was made empirically based on clustering results for datasets that were confirmed to feature Web attacks.

To find the optimum number of clusters, we express the validity of clustering results in terms of an 'internal index' after every HCA step. This index indicates how well observations, i.e., histograms, lie within their cluster, and how well clusters are distanced. As opposed to an external index, an internal index does not require external information (e.g., ground truth) for validation, which is desirable in our case because we need to account for diverse datasets. As shown in [17], a cluster can be graphically represented by its so-called *Silhouette*, which is composed of the Silhouette coefficients of each observation in that cluster.

Silhouette coefficients are mathematically defined as follows:

$$s(x) = \frac{b(x) - a(x)}{max(a(x), b(x))} \tag{5.5}$$

In our context, $a(x)$ represents the average dissimilarity of histogram X to all other objects within the same cluster, while $b(x)$ is the inter-distance of histogram X to its neighboring cluster (i.e., the second-best cluster choice for histogram X). Silhouette coefficients are particularly interesting for our approach, since they can be calculated based only on pair-wise distances between observations in the dataset, for which we can use the MDPA metric. The average of all Silhouette coefficients lends itself well as an internal index, and as such can be used to validate clustering results and determine the optimum number of clusters [17]. To this end we have also studied alternative internal indices, such as the one used in Calinski & Harabasz (CH) [4], which is based on a 'sum-of-squares'. Unlike the average Silhouette coefficient, the CH index is based on the distance of the cluster centroids to the general mean of the data [14]. Since the MDPA metric does not allow for the mean between more than two histograms to be determined, neither cluster centroids, nor the general mean can be calculated. This yields not only the CH approach infeasible in combination with MDPA, but any other method that relies on 'sums-of-squares'.

5.3.4 *Brute-force* Phase Detection

Detection of the *brute-force* phase is always done using the largest cluster of a tuple, since we assume that attack traffic is dominant enough to comprise a (large) cluster by itself. In cases of non-attack traffic, the largest cluster may however contain histograms that are not very similar, meaning that distances between histograms are rather large. To filter out such candidates, we calculate the average intra-cluster distance for the largest cluster. In case it exceeds a predefined threshold θ,[4] we ignore the tuple in the remainder of the detection procedure.

Several CMSs, such as recent versions of Joomla and Drupal, have built-in mechanisms to mitigate simple brute-force attacks against their backends, mostly by requiring a token, served by the CMS as part of a session cookie or a form nonce, to be included in authentication requests. Depending on the attacked CMS, the token(s) required for authentication may have to be retrieved only once per attack (Joomla) or once per authentication request (Drupal). The thought behind this is that not-so-clever brute-force tools start their attacks without retrieving the authentication pages first, causing the attacks to never yield any useful result. However, our analysis of modern attack tools has revealed that they

[4]We use $\theta = 3$, meaning that a cluster's histograms must be roughly identical, i.e., three bytes deviation on average. This value was empirically established based on the analysis of real attacks.

are perfectly able to circumvent this type of protection nowadays, which will be reflected in the overall cluster structure. Therefore, besides analyzing merely the largest cluster, we analyze the relation between the two largest clusters and verify whether they have the following characteristics:

- Clusters must feature a typical relation in terms of size, such as $1:1$, $1:2$ or $1:3$.
- The average distance between histograms in both clusters, γ, must be at least 75. This value has been established empirically based on our datasets and our analysis has shown that many CMSs will trigger such behavior with a distance of only around 200. This distance can be explained by the completely different nature of requests for token retrieval and authentication attempts. Also, this minimum is used to avoid considering two clusters that have somewhat similar histograms which should not have been separated into these two clusters to begin with.

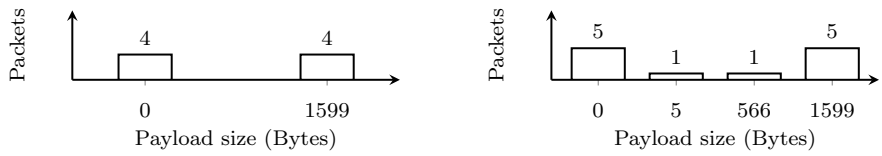
Given the alternating nature of token retrieval (typically done using HTTP GET requests) and authentication attempts (using HTTP POST), we refer to this behavior as *GET/POST-alternation* or *GET/GET/POST-sequence*.

Finally, once the largest cluster is found to feature a *brute-force* phase, we perform a sanity check to rule out false positives: in case the set of clusters features many small clusters, i.e., clusters with only one or two histograms, we 'overrule' the detection of the *brute-force* phase. Many small clusters indicate that the network traffic was highly variable in terms of payload, therefore contradicting our definition of typical attack behavior.

5.3.5 *Compromise* Phase Detection

Authentication attempts that ultimately result in a compromise are no different from connections resulting in failed authentication attempts, as the request sent to the Web server is basically identical (except for the credentials themselves). We therefore analyze return traffic, i.e., traffic from target to attacker, to identify potential compromises. Our analysis of attacks has shown that these return connections are different in size upon successful authentication. The difference in size can be explained by the (new) page, e.g., CMS backend panel, that is served to the attacker, which is different from the login forms used in the *brute-force* phase.

Since compromises can only be present after login attempts, an attack must reside in the *brute-force* phase before the *compromise* phase detection is activated. To detect compromises, we retrieve flow data from target to attacker and cluster the payload histograms in exactly the same way as for the *brute-force* phase detection. Since traffic from target to attacker consists of many authentication errors in a typical brute-force attack, we assume that traffic to be rather alike in terms of its payload distribution. If there exists only one cluster with a single



(a) *Brute-force* phase (b) *Compromise* phase

Figure 5.8: Histograms for flows in various phases of an attack against a vanilla Drupal instance.

histogram (referred to as the *single-histogram cluster* in the remainder of this section), we continue the detection. Otherwise, many small clusters point to traffic that is scattered in terms of payload, while clusters with more histograms are unlikely to feature a compromise, because the compromise should be considered an 'exceptional' case within the attack.

Based on measurements in a lab environment, where we have recorded the network traffic between the attack tool *Patator* and the three CMSs considered in this work, we have identified various characteristics of a compromise. As such, if a histogram in the single-histogram cluster matches the following two criteria, we consider it a compromise:

- The histogram 'size', i.e., each bin multiplied by its value, is larger than the average size of all other clusters.
- The histogram's bins must differ from bins in all other clusters, since we assume that the flow for the compromise carries significantly different traffic, such as a CMS backend management page. The only exceptions are the zero-value bins, which are ignored from our detection, as explained in Section 5.3.3, and the bins that represent the Maximum Segment Size (MSS), as they are also likely present in both *brute-force* phase and *compromise* phase histograms.

These characteristics are visualized in Figure 5.8, where we show both a *brute-force* phase histogram and a *compromise* phase histogram based on our measurements. It is clear that the overall size of the *compromise* phase histogram is larger than the size of the *brute-force* phase histogram and that the histogram's bins are different. It should be noted that the *brute-force* phase obviously consists of a whole bunch of histograms like the one shown in Figure 5.8a, rather than only one.

5.4 Validation

In the context of our validation, we define an *attack* as a sequence of at least N flows towards a CMS backend. Due to the fact that connections that are

part of an attack feature at least one authentication attempt, N consecutive flows feature at least N authentication attempts. By measuring the number of consecutive connections towards the same CMS backend Uniform Resource Locator (URL), we have found that $N = 20$ covers roughly the upper 10% of attack sizes measured in our validation datasets. This value, on the one hand, causes benign failed login attempts to be filtered out implicitly, and, on the other hand, reduces the load on our prototype due to very small attacks and noise.

The remainder of this section is structured as follows. In Section 5.4.1, we introduce the two prototypes that are developed for demonstrating the contributions of this work. Then, in Section 5.4.2, we explain the datasets that have been collected and analyzed for validation. This is followed by an introduction of our validation approach and the applied metrics in Section 5.4.3 and 5.4.4, respectively, followed by a discussion of the validation results in Section 5.4.5.

5.4.1 Prototypes

For the sake of validating our detection approach, we have implemented two prototypes. First, we have modified the IPFIX Metering and Exporting processes of our flow exporter, such that it is able to export payload size histograms for every observed/metered flow using IPFIX. INVEA-TECH's FlowMon platform was chosen for this purpose, as it has been designed with extensibility in mind. Nevertheless, our extension can easily be ported to other flow exporters, such as nProbe[5] and YAF[6]. Second, we have implemented an intrusion detection prototype that performs the preselection and detection as described in Section 5.4.3, which is available on GitHub.[7]

5.4.2 Datasets

The datasets used for validation of this work have been collected in the network of Hosting 2GO, a Dutch Top-10 Web-hosting provider, for a period of a month in July/August 2015. The systems under observation host approximately 2500 Web hosting accounts, worth a total of 2603 *vhosts*. In total, the datasets consist of traffic records worth 414 GB, generated by more than $237k$ hosts. More specifically, we use two types of datasets, summarized in Table 5.1, both from the same observation point, but collected on different systems:

- *Log files* – These Web server access log files serve as ground-truth for our validation and are the only means of verifying whether an attacker has compromised a system.
- *Flow data* – This data has been exported using IPFIX with $1:1$ sampling applied on the exporting device, and consists of the typical set of fields seen

[5] `http://www.ntop.org/products/netflow/nprobe/`
[6] `https://tools.netsa.cert.org/yaf/`
[7] `https://github.com/ut-dacs/https-ids`

Dataset	Size on disk	Size (entries)
Access logs	2.8 GB	12.2 M
Flow data	16.8 GB	42.1 M

Table 5.1: Characteristics of validation datasets.

in many NetFlow v9 implementations. Among those fields are IP addresses and port numbers, L3 protocol number, IP Type of Service (ToS) byte and SNMP input interface ID. Additionally, we augment the exported data with HTTP information, namely hostname and URL, and payload metadata, as discussed in Section 5.2. The functionality for parsing hostnames in HTTP and HTTPS traffic, and URLs in HTTP traffic is available on every modern FlowMon device. The code for exporting payload size histograms is our own. Our flow data serves as input for the prototype described in Section 5.4.1.

To protect the privacy of individuals of whom we have captured network traffic and log files, we have anonymized all fields that can potentially lead to personal identification. First and foremost, we have anonymized all IP addresses in both the flow data and log files in a *prefix-preserving* manner using the de facto standard in this area: Crypto-PAn. Prefix-preserving in this context means that if two IP addresses share a k-bit prefix in the non-anonymized dataset, their anonymized counterparts will do so as well. Also, Crypto-PAn is consistent across traces, such that we can correlate IP addresses in both datasets, even after anonymization. Second, we have hashed all HTTP hostnames (i.e., *vhosts*), such that the original hostname cannot be retrieved anymore, while hostnames can still be uniquely identified in both datasets.

5.4.3 Approach

Out of the three brute-force attack phases discussed in Chapter 4, only the *scan* phase has not been touched in this work, due to its irrelevance in the context of attacks against Web applications. We therefore validate our detection performance only with respect to the *brute-force* and *compromise* phases. Additionally, to underline the improvements of our approach compared to our elementary approach, i.e., the state-of-the-art in the area of flow-based brute-force attack detection for Web applications, we also compare our results to results obtained based on the approach described in [71].

Before we can compare detection results to our ground-truth, we have to post-process the datasets to obtain the same unit of comparison, as shown in Figure 5.9. For this purpose, we define an *interaction* as a set of consecutive sessions/connections towards a CMS backend within a certain time period. To obtain interactions from Web server log files, we developed an Apache access

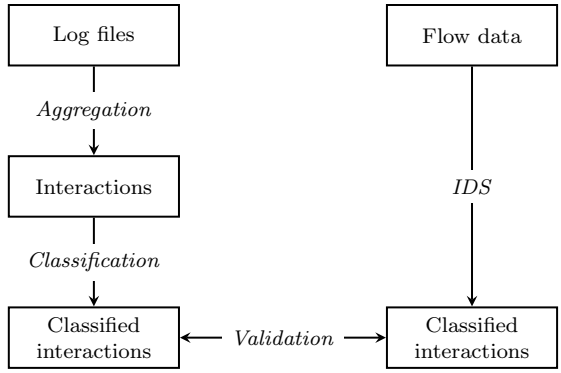


Figure 5.9: Dataset post-processing for validation.

log parser[8] that aggregates log records into interactions based on a number of heuristics. In the case of flow data, which naturally consists of session/connection entries, the only post-processing needed is the aggregation of sessions between a tuple with less than 5 minutes of idle time between sessions. This idle time was chosen as a tradeoff between the typical, very short-lived HTTP(S) sessions between server and client on the one hand, and a buffer for coping with time offsets between datasets on the other. After post-processing, our ground-truth consists of 854,945 interactions.

As for identifying brute-force attacks and compromises from log files, we take the following approaches. For the *brute-force* phase, we determine the number of consecutive HTTP POST requests towards CMS backend login pages, listed in Appendix B. If this number exceeds N, we classify the interaction as malicious. For the *compromise* phase, the general approach is to label an interaction as to feature a compromise as soon as an attacker is not interacting with the login form anymore, which however strongly varies per CMSs. Wordpress is the most simple case, since it uses different URLs for login page and backend. Joomla's backend page after login is greater in size than the login page itself, so one or few larger responses by the Web server signal a login. In the case of Drupal, a successful login is indicated by a different HTTP status code, which performs a redirection and thus carries less HTTP payload. An overview of these behaviors for the various CMSs is provided in Appendix B.

5.4.4 Metrics

IDS performance metrics are typically expressed in terms of positive and negative detections being either true or false. We therefore define metrics for the *brute-force* phase as follows:

[8] `https://github.com/ut-dacs/usenix_sec16`

- TP_B: Interaction labeled as malicious that is reported by our prototype.
- FP_B: Interaction labeled as benign that is reported by our prototype.
- TN_B: Interaction labeled as benign that is not reported by our prototype.
- FN_B: Interaction labeled as malicious that is not reported by our prototype.

These metrics can also be expressed as percentages. For example, the TPR is defined in the context of this work as the percentage of interactions correctly labeled as malicious and reported by our prototype:

$$TPR = \frac{TP}{TP + FN} \tag{5.6}$$

To avoid any bias of *brute-force* phase detection results on detection results for the *compromise* phase, we only consider those attacks that were successfully found to feature a *brute-force* phase. This is a logical consequence of the fact that the *compromise* phase can only be reached after the *brute-force* phase, as described in Chapter 4. From the ground truth, we conclude a compromise upon change of a URL between a tuple from a backend authentication URL to another URL on the same *vhost*, after more than N authentication attempts. As for our prototype, we measure its performance in terms of the following evaluation metrics for the *compromise* phase:

- TP_C: TP_B correctly identified to feature a compromise.
- FP_C: TP_B incorrectly identified to feature a compromise.
- TN_C: TP_B correctly identified to not feature a compromise.
- FN_C: TP_B incorrectly identified to not feature a compromise.

Additionally, we use the aforementioned evaluation metrics to calculate the accuracy (*Acc*) of our prototype, as previously defined in 4.1.

We have found some (positive and negative) detection results for the *brute-force* phase that could not be matched with our ground-truth, mostly because of timing deviations between our datasets. To make sure that these detections are not accounted wrongly in any of our evaluation metrics, we have listed them separately as *unclassified*.

5.4.5 Results

Our validation results are shown in Table 5.2, which lists the values for all evaluation metrics, as well as their respective rates/percentages. The most important take-away is that our approach (*new*) significantly outperforms our elementary approach (*old*), in all respects. Besides detecting a significantly larger number of

Phase	Type	TP	FP	TN	FN	*Acc*	*Unclassified*
Brute-force	New	469 (0.597)	519 (0.000)	1,748,486 (1.000)	317 (0.403)	1.000	21,336
Compromise		1 (1.000)	14 (0.030)	454 (0.970)	0 (0.000)	0.970	0
Brute-force	Old	237 (0.303)	5,058 (0.003)	1,743,904 (0.997)	545 (0.697)	0.997	21,382
Compromise		–	–	–	–	–	–

Table 5.2: Validation results.

TP_B (469 vs. 237), the number of false detections has been reduced to almost 10% of existing works. Also, our approach has shown to be able to detect the one compromise in the dataset. It should be noted that we validated our work in a conservative case, because our validation network has a firewall in place that blocks remote hosts when generating too many connections. Consequently, large attacks have never reached the Web servers and are therefore not recorded in our datasets, while they would likely be detected by our prototype.

With respect to false detections, we made several observations. First, almost all FPs are somehow related to photo galleries. Photos in a gallery are often similar in size, because they have been shot by the same camera and post-processed in the same way. They may even have been compressed such that they end up having the same size. Thumbnails are even worse (for our approach); Once an album is opened, thumbnails of the album's contents are fetched by the client, resulting in tons of semi-identical connections, due to the fact that the thumbnails have exactly the same dimensions and are typically identical in size. Second, FNs have two major causes: either they are caused by low-intensity attacks that do not generate more than N connections/requests per five minute data chunk, or the distance between histograms in the largest cluster slightly exceeds our threshold θ.

5.5 Conclusions

In this chapter, we have shown how to perform compromise detection for Web applications using flow data, in a way that is decoupled from the hosting infrastructure and works in encrypted environments. Our detection approach based on the use of histograms together with clustering methods allows us to overcome the problems with TCP phenomena like retransmissions and control information, and false positives as a consequence of benign applications generating many similar connections, such as Web crawlers (Section 5.2). In addition, it allows us to discriminate between attack traffic and non-attack traffic makes our approach applicable to almost any Web application (Section 5.3).

For validating our detection algorithm, we have selected an observation point that hosts many Web applications in a realistic environment: a Dutch Top-10 Web hosting provider. Our monthlong validation, described in Section 5.4, with a prototype implementation of our detection algorithm has shown to be able to detect the one compromise in the dataset. In addition, it has proven that our approach outperforms our elementary approach (described in [71]) in all respects. For example, the number of true positives is doubled while the number of false detections has been reduced to almost 10% of the original value. Given that promising results have been achieved based on real deployments in a large Web hosting environment and the fact that our prototype may be used by Web hosting providers as part of a two-step blocking process, we conclude that flow-based compromise detection for Web applications is not only viable, but may even see production deployment in the near future.

Based on the differences in the detection results between compromise detection for SSH, described in Chapter 4, and compromise detection for Web applications, we conclude that it is less difficult to perform compromise detection for applications and protocols that have an authentication restriction than for applications and protocols that have not. SSH is a typical protocol that features such a restriction, since successful authentication is required to perform any action on the remote machine. Web applications however do not feature this restriction, as attackers may perform any other action between authentication attempts. As a result, attacks against protocols like SSH provide a more significative behavior than attacks against Web applications.

Part III

Resilient Detection

CHAPTER 6

Resilient Detection

The foregoing chapters have discussed how to perform compromise detection for SSH (Chapter 4) and Web applications (Chapter 5) using flow data. By their design, flow caches in export devices are accounting every individual connection, which makes flow monitoring systems however susceptible to traffic mixes that consist of many small connections. Examples of such mixes are large network scans (e.g., as part of brute-force attacks) and flooding attacks. Consequently, they may be used to effectively blind flow monitoring systems, since flow caches are overfull with attack flow records.

Although flooding attacks are not the main target of this thesis, we take them as an extreme case to demonstrate that flow monitoring systems are susceptible to overload by many small connections in the network. We start by investigating which components of flow monitoring systems are susceptible to flooding attacks, as well as (1) how we can detect them and (2) eventually detect overload of the system as a whole. We do this by measuring attacks in a network where many large ones are observed per day: the Czech NREN CESNET. After understanding the important characteristics of flooding attacks, we develop and validate a lightweight detection module that provides a means to filter attack traffic in such a way that the resilience of the monitoring system is retained. Key is the move of the detection module from the flow collector – where it would normally reside – towards the data source: the flow exporter. The contribution of this chapter are algorithms and prototypes for detecting flooding attacks by both dedicated and embedded flow export devices.

The papers related to this chapter are [48], [69].

The organization of this chapter is as follows:

- *Section 6.1 provides background information on this chapter and states its contribution.*
- *Section 6.2 describes flow-level characteristics of flooding attacks.*
- *Section 6.3 elaborates on existing flow attack detection algorithms.*
- *Section 6.4 explains the validation setup and results.*
- *Section 6.5 elaborates on the feasibility of the work presented in this chapter in terms of deployability on various high-end packet forwarding platforms.*
- *Section 6.6 concludes this chapter.*

6.1 Background & Contribution

One of the main tasks of flow exporters is to aggregate individual packets into flows, such that network traffic can be exported in a more scalable fashion compared to regular packet capture. The achieved scalability gain comes from the fact that flows, which are meant to resemble connections, consist of multiple packets and only one or few flow records are exported per flow. As such, only a minor fraction of the original traffic volume – in the order of 1/2000, as shown in Chapter 2 – is actually being exported. However, as soon as flows consist of only few packets or even only one, the scalability gain vanishes completely. Moreover, depending on the nature of the traffic, flow monitoring may even result in traffic amplification once the exported traffic becomes larger in terms of packets and bytes than the original traffic.

Also another disadvantage of the use of flow monitoring can be identified, besides the problem of decreased resiliency. Compared to regular packet capture, flow-based analysis applications and appliances, such as IDSs, are subject to delays during flow metering, export and collection [47], due to the design of NetFlow and IPFIX. These delays are in particular a consequence of timeouts for expiring flow records as part of the flow metering process (Chapter 2, and processing times of flow collectors. Considering the default idle timeouts applied by vendors and the processing times of popular flow collectors, this usually results in IDS detection delays in the order of minutes.

In situations where the monitoring infrastructure is under attack, it is important to detect and mitigate as early as possible to limit the potential damage, such as device overload and link capacity exhaustion. Our intuition tells us that moving parts of the detection process closer to the data source may reduce detection times drastically. Given that a timely detection allows for timely mitigation, we propose a functional extension for flow exporters that integrates intrusion detection into the flow metering process. We do this in the context of DDoS (flooding) attacks, given that it is widely known that flow monitoring systems are susceptible to this type of attack. Our approach avoids the delays incurred in typical flow monitoring systems and has the following advantages:

1. Mitigation of DDoS flooding attacks by filtering malicious flow data before it reaches and potentially overloads a flow collector (as illustrated by (1) in Figure 6.1). We know from our operational experience that DDoS attacks often cause flow data loss due to collector overload [64]. Moreover, European backbone operators have also confirmed this problem in discussions we had with them.

2. Mitigation of DDoS flooding attacks by blocking malicious traffic before it reaches the Local Area Network (LAN) (as illustrated by (2) in Figure 6.1).

Typical values for the idle timeout applied for expiring flow records range from 15 seconds (default value applied in Cisco IOS [155]) to 60 seconds (default value

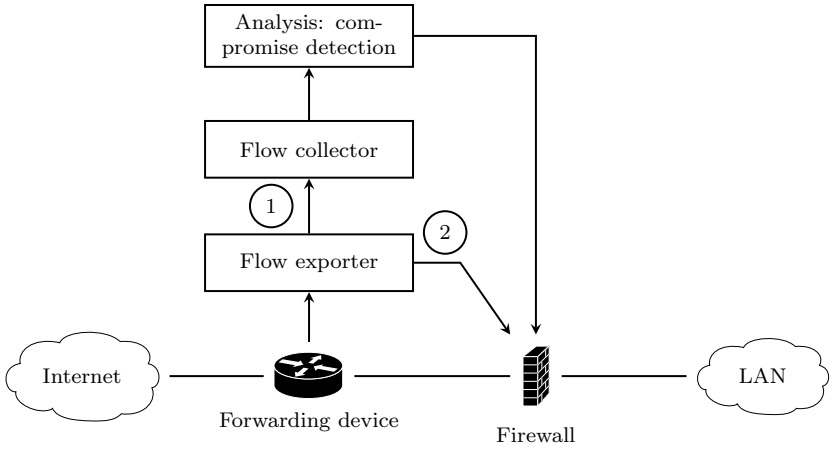
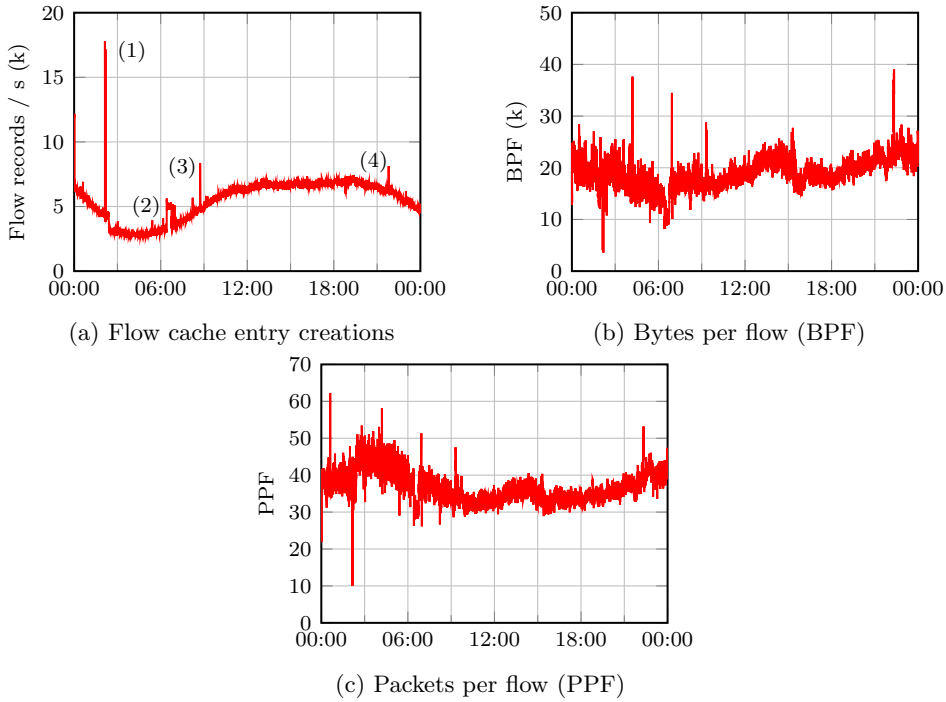


Figure 6.1: Typical flow monitoring system deployment.

applied in Juniper Junos [131]). In addition, flow collectors often work based on time slots, which causes flow data to become available to analysis application only after the next time slot has started. For example, the popular flow collector *NfSen* uses time slots of 5 minutes, resulting in an average delay of 150 seconds (2.5 minutes). The average delay between the moment in which a packet is metered and the time at which flow data is made available to analysis applications, is therefore at least 165 seconds, considering an idle timeout of 15 seconds. In this chapter, we analyze whether our approach can reduce the delay up to 10% of this value, such that negative effects of flooding attacks on the monitoring infrastructure can be mitigated as early as possible. Besides this requirement, we target an intrusion detection module that is lightweight, having a minimal performance footprint of 10% in terms of CPU usage and memory consumption on a flow exporter. This is important since exporter operation is considered time-critical. Last, the accuracy of our intrusion detection module should be high enough, to ascertain a low number of false positives/negatives.

6.2 DDoS Attack Metrics

Flooding attacks are a type of (D)DoS attack that aim at exhausting targets' resources by overloading them with large amounts of traffic or (incomplete) connection attempts. As every connection attempt uses a different source port number and is therefore a flow on its own, large numbers of flow records are exported to flow collectors, effectively canceling out the data aggregation advantage provided by flow export technologies. In case a target replies to a connection attempt, two flow records are exported per attempt. The same characteristics may apply



(a) Flow cache entry creations

(b) Bytes per flow (BPF)

(c) Packets per flow (PPF)

Figure 6.2: Flow-based DDoS attack metrics.

to (large) network scans. Flow collectors need to process all resulting records, consisting of only a few packets and bytes, which may be more than they can handle.

In this section, we analyze which flow-level traffic metrics are suitable for lightweight detection of attacks on a flow exporter. In [64], four traffic metrics were identified that change significantly during a (D)DoS flooding attack: flow cache entry creations per second, average flow duration, average number of bytes per flow, and average number of PPF. All but the average flow duration can be monitored on a flow exporter by using only counters, without the need to access and process each individual flow record after expiration. This makes these metrics particularly interesting for this work, in which we aim at designing a lightweight intrusion detection module for detecting large flooding attacks.

Time-series of the three considered metrics are shown in Figure 6.2. The subfigures show data from one of the backbone links of the Czech national research and education network CESNET in November 2012. The number of flow cache entry creations per second is shown in Figure 6.2a. The diurnal pattern is clearly identifiable and several peaks can be observed. Given the flow-level characteristics of flooding attacks, we assume the peaks labeled with a number to indicate

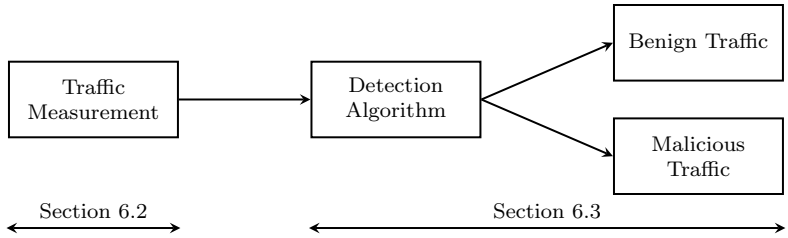


Figure 6.3: Detection workflow.

the presence of such attacks. We have validated this assumption by manually verifying the presence of scanning and flooding attacks in the flow data.

The number of Bytes per Flow (BPF) and PPF are shown in Figure 6.2b and Figure 6.2c, respectively. Although the attacks identified in Figure 6.2a can be observed as negative peaks here as well, the figures also show many other peaks, which make these metrics noisy. What can be confirmed from these figures, however, is that the attacks identified in Figure 6.2a consist of many small flows with few and small packets.

Out of the three presented metrics in Figure 6.2, the number of flow cache entry creations (Figure 6.2a) appears to be the most suitable metric for our purposes for the following reasons. First, it shows the least amount of noise, peaks are clearly identifiable, and the identified peaks have been confirmed to be attacks, as substantiated by the other metrics. Second, this metric is the best to fulfill the requirement of being lightweight, as only a single counter is needed that has to be reset after every measurement interval.

Although we have shown only day-long time-series in Figure 6.2, we have verified that our conclusions are valid for the whole dataset. We will therefore use the number of flow cache entry creations for our traffic measurements, as shown in Figure 6.3. These measurements are performed in a continuous fashion and used as input for a detection algorithm, which on its turn classifies a measurement (sample) as benign or malicious. Two detection algorithms are discussed in the next section.

6.3 Detection Algorithms

In this chapter we consider an anomaly-based intrusion detection approach. One method for performing anomaly detection based on the analysis of time-series is forecasting, which uses previous measurements for forecasting the next value. If the measured value does not lie within a certain range of the forecasted value, a measurement sample is considered malicious and an anomaly has been detected. We consider the following two algorithms:

Algorithm 1: Exponentially Weighted Moving Average (EWMA) for mean calculation, extended by thresholds and a CUmulative SUM (CUSUM) [19]. We consider this our basic algorithm.

Algorithm 2: Algorithm 1, extended by seasonality modeling.

Although the second algorithm may intuitively be considered more accurate, it is likely to have a larger performance footprint in terms of memory consumption and processing complexity than the first algorithm, since more data needs to be stored and processed. Whether the larger performance footprint justifies the use of Algorithm 2 in terms of accuracy, will be investigated later in this chapter.

Both algorithms rely on EWMA for calculating the mean over past values, which we use for forecasting the next value. Previous works have shown that EWMA can be used for anomaly detection (e.g., [19], [23]). It is defined as follows:

$$\bar{x}_t = \alpha \cdot x_t + (1-\alpha) \cdot \bar{x}_{t-1} \tag{6.1}$$

$$\hat{x}_{t+1} = \bar{x}_t \ , \tag{6.2}$$

where x_t is the measured value, $\bar{x}_t$ is the weighted mean over current and past values at time t, $\hat{x}_{t+1}$ the value forecasted for time $t+1$, and $\alpha \in (0,1)$ a parameter which determines the rate in which previous values are discarded. When the value of x_t becomes known, both the forecasting error e_t and an upper threshold $T_{upper,t}$ can be calculated:

$$e_t = x_t - \hat{x}_t \tag{6.3}$$

$$T_{upper,t} = \hat{x}_t + max(c_{threshold} \cdot \sigma_{e,t}, M_{min}) \ , \tag{6.4}$$

where $c_{threshold}$ is a constant and $\sigma_{e,t}$ the standard deviation of previous forecasting errors. M_{min} is a margin that is added to the measurement $\hat{x}_t$ to avoid instability in case $c_{threshold} \cdot \sigma_{e,t}$ is small. This solution prevents small peaks during quiet periods to be considered anomalous. Note that we do only consider an upper threshold and no lower threshold, since flooding attacks result, by definition, in a greater input value in terms of flow record creations than the forecasted value (as discussed in Section 6.2).

Reporting an anomaly every time the upper threshold has been exceeded may result in a large number of false positives. To overcome this problem, we use a CUSUM, which is widely used in anomaly detection algorithms [19], [58], [76]. The differences between the measurement and the upper threshold are summed (S_t), and an anomaly is detected when the sum exceeds threshold $T_{cusum,t}$:

$$S_t = max(S_{t-1} + (x_t - T_{upper,t}), 0) \tag{6.5}$$

$$T_{cusum,t} = c_{cusum} \cdot \sigma_{e,t} \ , \tag{6.6}$$

where c_{cusum} is a constant. A measurement is flagged anomalous every time $T_{cusum,t}$ has been exceeded. To improve the precision of anomaly end time detection, we use an upper bound on S_t to let S_t decrease faster after x_t has decreased.

The presented algorithm relies only on the forecasted value and the measured value. Due to the daily periodicity of network traffic and the quick rises and falls of network utilization during mornings and evenings, respectively, the detection algorithm may benefit from a longer history. Our second considered detection algorithm uses the Holt-Winters Additive forecasting method for modeling seasonal components [31]: By adding a linear and a seasonal component to a base signal, the next value is forecasted. As a consequence of the daily periodicity of network traffic, we use day-long seasons. Since we do not identify a significant linear trend in network traffic at this timescale, we disregard the linear component. The weighted mean of the base component is calculated based on previous measurements on the same day, while the mean of the seasonal component is calculated over values at the same time in previous days. We define these two components as follows:

$$b_t = \alpha \cdot (x_t - s_{t-m}) + (1 - \alpha) \cdot b_{t-1} \tag{6.7}$$

$$s_t = \gamma \cdot (x_t - b_t) + (1 - \gamma) \cdot s_{t-m} \tag{6.8}$$

$$\hat{x}_{t+1} = b_t + s_t \ , \tag{6.9}$$

where b_t and s_t are the *base* and *seasonal* components of the forecasted value $\hat{x}_{t+1}$, respectively, m is the season length (i.e., number of measurement intervals per day, since network traffic shows daily periodicity), and $\gamma \in (0, 1)$ a parameter which determines the rate in which previous values are discarded. The previous values in this case are not from the previous measurement interval, but from the same interval in the previous season (i.e., day). The initial base value is set to the average of all measurement values in the first season. Therefore, a training period of one season is needed.

The use of day-long seasons and small measurement intervals results in a large number of measurement values per season. To support our requirement of being lightweight, we only store seasonal values every hour and interpolate between those. This also reduces measurement noise, which would otherwise imprint in the seasonal values. Besides that, precautions need to be taken to not let measurements during anomalies influence the forecasts, which can be accomplished by not updating s_t, b_t and e_t during an anomaly. This is because we aim at forecasting non-anomalous network behavior. Another improvement made to the algorithm is to separate algorithm states for weekdays and weekends, since the traffic behavior usually varies significantly between these types of days. As such, forecasting of weekend days is done based on the traffic behavior of the previous weekend, instead of working days. Analogously for weekdays.

It has been shown in [19] that an interval between 5 and 20 seconds yields best results for detecting flooding attacks using the CUSUM method. Our mea-

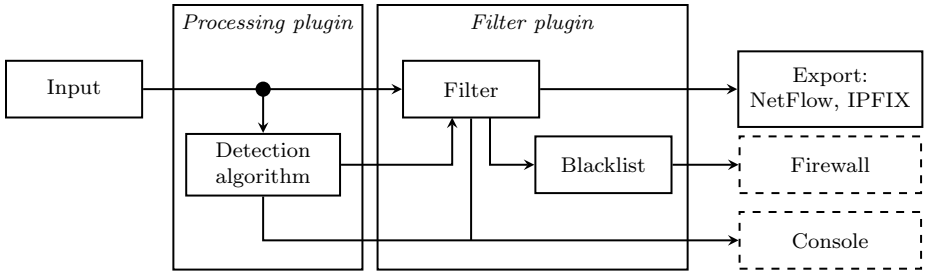


Figure 6.4: Prototype architecture.

surements have shown that an interval of 5 seconds indeed results in the most accurate detection results, but an extensive comparison of various interval lengths is out of the scope of this chapter.

6.4 Validation

This section describes the setup used for validating our work, as well as the validation results. We start by discussing the developed prototype in Section 6.4.1, after which we provide details on the two datasets used for validating our requirements in Section 6.4.2. Then, in Section 6.4.2.1, we perform the actual validation based on the requirements identified in Section 6.1.

6.4.1 Prototype

Our prototype implements both the traffic measurements based on the metric chosen in Section 6.2 (i.e., the number of flow cache entry creations) and the detection algorithms presented in Section 6.3. It is developed as a plugin for INVEA-TECH's FlowMon platform, which we selected both because we have full control over it in our networks, and because of its highly customizable architecture based on plugins for data input, flow record processing, filtering, and export. The prototype is designed as a hybrid processing and filtering plugin. Default input and export plugins from INVEA-TECH are used for packet capturing and NetFlow and IPFIX data export. Information gathered by the prototype, such as detected anomalies, is sent to a console. The complete architecture is depicted in Figure 6.4.

The intrusion detection module is implemented as a processing plugin. After every measurement interval, the algorithm is run and the measurement sample is classified as benign or malicious. The result is then passed on to a filter plugin, which is used for attack mitigation. When measurement samples are classified as benign, the corresponding flow records are passed on to the export plugin. Otherwise, the filter plugin identifies attackers as soon as an attack

	Dataset 1	Dataset 2
Duration	14 days	10 days
Period	August/September 2012	October/November 2012
Flows	10.0 G (717.0 M per day)	6.7 G (668.9 M per day)
Packets	257.1 G (18.4 G per day)	186.7 G (18.7 G per day)
Bytes	134.5 T (9.6 T per day)	128.5 T (12.8 T per day)
Anomalies	131	11
Anomaly duration	Minimum: 5s Average: 5m, 55s Maximum: 2h, 41m, 50s	Minimum: 5s Average: 15m, 55s Maximum: 2h, 48m, 55s

Table 6.1: Dataset compositions.

has been detected. Attackers are identified by counting the number of exported flow records per source IP address. When more than F flow records per second with less than 3 packets and identical source IP address have been exported, the address is added to a *blacklist*. Measurements have shown that $F = 200$ is high enough to ascertain that a blacklisted host was flooding a network or host, and that benign hosts should never become blacklisted. In the case of attacks with spoofed IP addresses, one could also consider blacklisting destination addresses. We have measured the effects of this approach as well, but source address blacklisting has yielded slightly better results.

After identification of the attackers, the filter plugin performs two actions, which correspond to the contributions identified in Section 6.1:

1. Firewall rules are composed and sent to a firewall to block the attackers' traffic. This corresponds to (1) in Figure 6.1.

2. Flow records with the attackers' IP addresses are filtered to reduce the stream of flow records sent to the collector. This corresponds to (2) in Figure 6.1.

When an anomaly has ended, the composed rule is removed from the firewall, counting of exported flow records is stopped, and all counters are reset. The filtering of flow records is stopped after T_{idle} seconds, where T_{idle} is the idle timeout of the flow exporter, to make sure that flow records in the exporter's *flow cache* that still belong to the attack are filtered.

6.4.2 Dataset

The dataset used for validating the detection algorithms has been captured on a backbone link of the CESNET network in August/September 2012. This link has a wire-speed of 10 Gbps with an average throughput of 3 Gbps during working

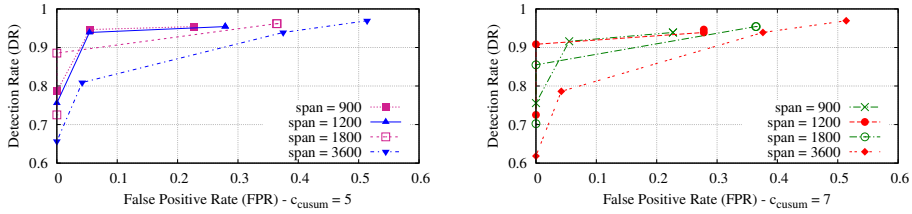


Figure 6.5: Receiver Operating Characteristics (ROC) of Algorithm 1.

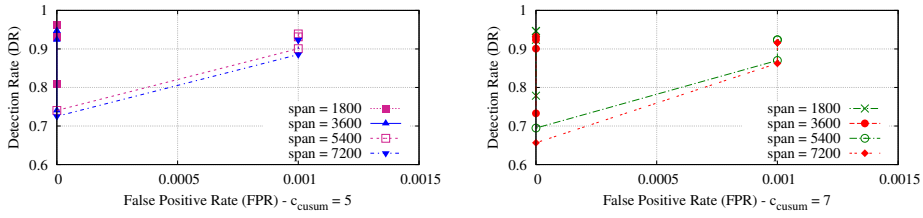


Figure 6.6: ROC of Algorithm 2.

hours. The dataset comprises 14 days of measurements, composed of the number of flow cache entry creations, packets and bytes per measurement interval (details are listed in Table 6.1, Dataset 1). To establish a ground truth for validation, we have manually identified anomalies that show a high intensity in the number of flow records. Samples belonging to an anomalous interval are labeled malicious. Other samples are labeled benign.

6.4.2.1 Results

In this section we validate whether our approach meets the requirements identified in Section 6.1. We start by validating the accuracy in Section 6.4.2.2, mainly because of the fact that an intrusion detection module with a poor accuracy would be useless in any setup. After choosing the algorithm that performs best in terms of accuracy, we validate the response time and performance footprint of this algorithm in Section 6.4.2.3 and Section 6.4.2.4, respectively.

6.4.2.2 Accuracy

The accuracy of both detection algorithms is visualized in Figure 6.5 and 6.6. The ROC curves show the impact of the constant $c_{threshold}$ on the Detection Rate (DR) and the False Positive Rate (FPR). The DR is a measure for the number

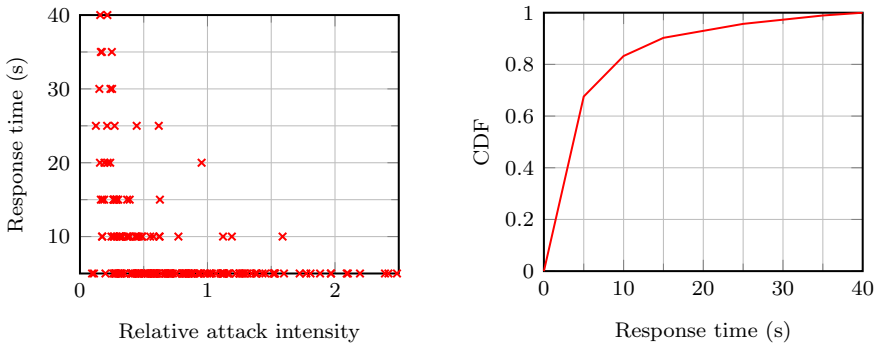
of attacks that have been detected correctly and is defined as follows [21]:

$$DR = \frac{\#\{\text{detected attacks}\}}{\#\{\text{attacks}\}} \tag{6.10}$$

The total number of attacks is determined by considering consecutive malicious samples to belong to the same attack. An anomaly is considered detected if approximately 50% of the samples is flagged malicious. The FPR is the ratio between the number of samples incorrectly flagged malicious and the number of samples labeled benign. In contrast to the more common practice of plotting the TPR (ratio between the number of samples correctly flagged malicious and the number of samples labeled malicious) versus the FPR, we plot the DR versus the FPR. This is because we do not require our algorithm to flag *all* samples of an anomaly, as long as the ones with a high intensity are catched. Each curve in the plots shows the accuracy for a different combination of *span* and c_{cusum}. *Span* represents the length in seconds of the history considered by the detection algorithm. As the algorithm is only aware of the number of measurement intervals and not of durations, we convert this time window to measurement intervals by dividing it by the length of a measurement interval. As such, it is used for calculating α ($\alpha = \frac{2}{N+1}$, where N is the number of intervals [23]) and for determining the number of values considered in calculating the standard deviation of forecasting errors $\sigma_{e,t}$. Besides *span* and c_{cusum}, all other parameters have been fixed: $M_{min} = 7000$ (Algorithm 1 and 2), $\gamma = 0.4$ (Algorithm 2).

Several observations can be made regarding the performance of Algorithm 1 in Figure 6.5. First, it is clear that the difference in c_{cusum} has little impact on the DR and FPR. Each pair of curves with the same *span* shows very similar growth. Second, increasing the *span* has little impact on the DR as well, but it increases the FPR significantly. This is because the forecast adapts slower to network traffic changes, such as diurnal patterns, and small deviations in the measurements are (incorrectly) flagged as malicious. Third, increasing $c_{threshold}$ affects the DR negatively: The highest DRs in the figure are achieved when the lowest $c_{threshold}$ is used. This is because the resulting higher upper threshold $T_{upper,t}$ will cause certain anomalies to stay below the threshold, resulting in a higher number of FNs. In the case of Figure 6.5, $c_{threshold} \in \{1.5, 2, 3, 5\}$. In our experiments, a *span* of 900 seconds and a $c_{threshold}$ of 3 yield the most optimal combination of a high DR (92%), while maintaining a low FPR (6%). In a typical deployment scenario as shown in Figure 6.1, however, this FPR is unacceptable as benign hosts may be blocked erroneously by our approach.

The ROC curve for various combinations of parameter values for Algorithm 2 is shown in Figure 6.6. Similar parameter values as for Algorithm 1 yield similar DRs, while the FPR is significantly lower, namely between 0% and 0.01%. Higher values of $c_{threshold}$ yield lower DRs, for the same reason as described for Algorithm 1. Again, $c_{threshold} \in \{1.5, 2, 3, 5\}$. When a *span* > 3600 seconds is used, the FPR increases slightly for small values of $c_{threshold}$, although still being very small (0.1%).



(a) Response times for various attack intensities.

(b) Cumulative Distribution Function (CDF) of the response time.

Figure 6.7: Response times of Algorithm 2.

In general, we conclude that Algorithm 2 is more suitable as a detection algorithm in our situation than Algorithm 1, since the FPRs are much lower while similar (high) DRs are maintained. We therefore conclude that the higher accuracy of this algorithm should excuse the additional performance footprint on the flow exporter. In the remainder of this section, we will therefore only consider Algorithm 2.

6.4.2.3 Response Time

The main objective of this chapter is to perform flow-based intrusion detection in near real-time. An important metric in the validation is therefore the response time. We define the response time as the time between the moment in which the algorithm detects an anomaly and the beginning of the anomaly. A scatter plot showing response times for various attack intensities is shown in Figure 6.7a, where we define the relative attack intensity as the fraction between the forecasting error (see (6.3)) and the forecasted number of flow records:

$$\frac{e_t}{\hat{x}_{t+1}} \tag{6.11}$$

The response time is always a multiple of 5 seconds, as this is the length of our measurement intervals. A response time of 5 seconds means that an anomaly has been detected within the same sample as the anomaly has started. As shown in the figure, most anomalies with a relative intensity larger than 0.3 are detected within 10 seconds. Outliers are the result of attacks that do not reach their full intensity right from the start. Anomalies with a relative intensity < 0.3 are mostly detected within 40 seconds. However, these anomalies are not the main target of our work as their potential damage to networks and hosts will

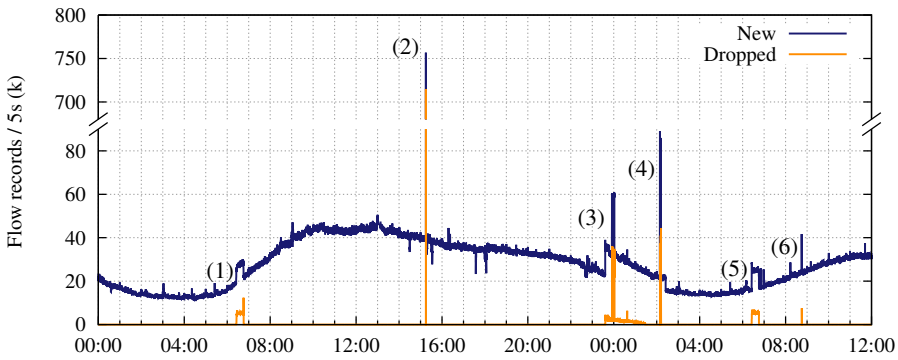


Figure 6.8: Prototype mitigation results.

be limited. Another view on the response times of the algorithm is shown in Figure 6.7b, where the CDF is plotted for each potential response time. It can be observed that 68% of all anomalies is detected within 5 seconds and 90% of the anomalies within 15 seconds. Note that these response times are even lower than typical idle timeouts of flow exporters, as explained in Section 6.1.

An example of the prototype in operation is shown in Figure 6.8. The figure shows the number of flow record creations, as measured by the processing plugin per measurement interval of 5 seconds, and the number of flow records dropped by the filter plugin per measurement interval, over a period of 36 hours. This measurement period is a subset of Dataset 2 (see Table 6.1). Several large anomalies can be identified, labeled as (1)-(6). The anomalies (1), (5) and (6) are clearly smaller than the others and are dropped largely or completely by the filter plugin. However, the main focus of our work is on very large anomalies, such as the anomalies marked as (2), (3) and (4). Anomaly (2) consists of 755k flow records per 5 seconds, while roughly 40k flow records have been forecasted. Out of these 755k flow records, our prototype is able to mitigate 715k. Anomaly (3) and (4) are both part of one longer anomaly, which is dropped partly throughout the duration of the attack. Anomaly (3) is mitigated completely, as the number of passed flow records roughly equals the forecasted number of flow records for measurement intervals during the attack (23k). Anomaly (4) is mitigated partially, where about 50% of the total number of flow records is dropped. When anomalies have not been dropped completely, one or more attackers generated less flows than the threshold of $F = 200$ per second. We do not consider this problematic since the number of passed flow records (40k per measurement interval) is in principle not causing collector overload, as this number equals the number of benign flow records at midday. Anomalies outside the visualized part of Dataset 2 (i.e., anomalies that are not shown in Figure 6.8) have been detected and mitigated similarly to the presented anomalies.

6.4.2.4 Performance Footprint

The last identified requirement for our intrusion detection module is that it is lightweight, as the operation of an exporter is considered time-critical. To verify whether our prototype fulfills this requirement, we have run the exporter process on our flow exporter both with and without our prototype. On average, the exporter process consumes less than 5% more CPU time when the processing and filter plugins are loaded. The memory footprint of the plugins depends on the size of an attack, as the number of flow records per IP address are counted under such circumstances. Our measurements have shown that the plugins never consume more than 20 MB of memory when the network is under attack.

6.5 Feasibility

So far we have used INVEA-TECH's FlowMon platform for validating all identified requirements for our intrusion detection module. This dedicated flow export platform has been chosen both because we have full control over it in our networks and because of its highly customizable architecture. An alternative approach would have been to use a high-end forwarding device with flow export capabilities.

Several platforms could have been chosen for implementing this, as long as they support some form of scripting for implementing the algorithm and retrieving statistics from the flow cache of the exporter. One option is Cisco's IOS, which supports scripting based on the Tool Command Language (TCL) as of version 12.3(2)T and 12.2(25)S. This provides administrators both a means to automate CLI command sequences and perform processing on information gathered from CLI commands and SNMP Management Information Bases (MIBs). Another option is Juniper's Junos, which is a specific command shell on top of a BSD-based kernel. It provides a full UNIX-level shell to administrators. Normal UNIX commands can therefore be used, which makes it straightforward to run customizations on a high-end device.

To measure whether our approach could work on any of these platforms, we have implemented it as an extension to one of the most widely deployed packet forwarding devices, the Cisco Catalyst 6500 [123]. In the remainder of this section, we elaborate on the various modifications to the detection algorithm that are necessary to implement the algorithm on Cisco's IOS.

6.5.1 Monitoring Information Available in IOS

Our detection algorithm is heavily based on a single metric, namely the number of flow cache entry creations per time interval. This metric is easily accessible on INVEA-TECH's FlowMon platform, since that platform has been designed with extendibility in mind. However, the amount of information available in IOS strongly depends on the path the packet or flow has taken within the router or

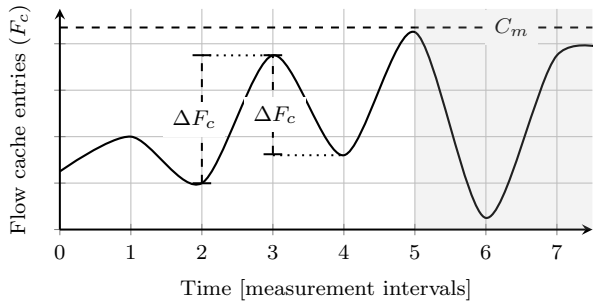


Figure 6.9: Flow cache entry creations in Cisco IOS over time.

switch. More precisely, packets are switched either in hardware or in software, although most packets are hardware-switched. On the campus network of the UT, for example, 99.6% of the traffic is hardware-switched, as explained in Section 3.4. Situations that trigger a packet to be switched in software are fragmented packets, packets destined to the forwarding device itself, and packets that require Address Resolution Protocol (ARP) resolution [118], for example. For flows processed in hardware, information on the number of flow cache entry creations is not directly available. To approximate this metric, we use the following information available from the flow metering and exporting process:

- Number of *flow cache entries* (F_c).
- Number of *exported software-switched flow records* (F_e).
- Number of *flow learn failures* (F_f). This metric is expressed in terms of packets, rather than flows.

The number of flow cache entry creations since the last measurement can be approximated using the following definition:

$$F = \Delta F_c + \Delta F_e + \frac{\Delta F_f}{c_f} \tag{6.12}$$

When flow cache entries are exported, F_c will decrease which will cause the approximation to be less accurate if the measurement intervals are too long. For example, in Figure 6.9, if the measurement were to cover two intervals, from $t = 2$ to $t = 4$, ΔF_c will not consider the peak at $t = 3$. By polling F_c more frequently, we can observe the changes more accurately, such that we observe the positive ΔF_c at $t = 3$ and the negative ΔF_c at $t = 4$, which is caused by exports. Then, if ΔF_c is negative, we use an estimation of previous ΔF_c values instead. When the flow cache is nearing its capacity limit, the exporter issues an emergency expiration, as explained in Section 2.4.3. In Figure 6.9 this is depicted

in the shaded area. As F_c reaches C_m, the flow cache capacity, most flow cache entries are expired. If a measurement is made between $t = 6$ and $t = 7$, the algorithm may detect this as an attack for one measurement interval, due to the vast increase in the numer of cache entries compared to $t = 6$. To counteract this, the implementation waits for the next measurement if it suspects an attack, to validate whether it is an actual attack. This does however increase the detection delay.

Since the number of entries in the flow cache (F_c) only regards hardware-switched flows, we also add the number of exported software-switched flows (F_e), which can be obtained directly from IOS. Finally, adding F_f allows for regarding flows that should have been created but were not, which is especially the case during high-intensity DDoS attacks, for example. To compensate for the fact that F_f is expressed in packets while the other metrics are expressed in flows, we divide F_f by the average number of PPF, represented by c_f in Equation 6.12.

6.5.2 Implementation

The Embedded Event Manager (EEM) – part of Cisco IOS that handles real-time network event detection – allows for the definition of *policies*, which can be used to execute an applet or script when events are triggered. For example, emails can be sent to network administrators when round-trip times reach a certain limit, or when network route changes occur. Another event type is based on time. This event can, among others, be scheduled at fixed time intervals. In this work, we use two time-based policies, implemented as TCL scripts:[1]

- **Measurement policy** – Determines the first component for our approximation of the flow-based metric: the number of flow cache entries (F_c), as described in Section 6.5.1.

- **Detection policy** – Retrieves the remaining components: the number of exported software flows (F_e) and the number of flow learn failures (F_f). Also, it implements the actual DDoS attack detection algorithm.

To obtain all three components, which are all made available using the SNMP protocol, we use a feature of the EEM environment that provides access to local SNMP MIB objects. The reason for splitting the measurement policy from the detection policy is that we require a higher resolution for the former to detect changes more accurately, as described in Section 6.5.1.

Policy invocations are memoryless, and since we want to share data – both between policy runs and between policies – a method for sharing data needs to be implemented. Due to the fact that the filesystem is flash-based, we generally want to avoid excessive write actions that will shorten the memory's lifespan. The

[1]The open-source TCL scripts can be retrieved from `https://github.com/ut-dacs/ios-ddos-detect/`

EEM environment therefore offers a *Context library* for this purpose; It allows for saving TCL variables to memory instead of writing them to disk. Besides for keeping track of our data between policy runs, we also use this feature to exchange information between the two policies, as the result of the measurement policy is needed by the detection policy.

The two policies discussed before are executed by the EEM at their respective intervals, which have been selected based on the runtime of the respective policies. When the switch is however under heavy load, its higher CPU utilization will cause the policies to take longer to execute. To avoid the policies from skipping an execution when the runtime of the policy exceeds the length of the interval, the prototype uses a feature from the EEM that can set a maximum policy runtime. If this runtime is exceeded, the policy terminates forcibly and data is lost. In the case of the detection policy, the algorithm has to start again from the learning phase as all state data is lost. If the measurement policy terminates prematurely, the measured number of created flow cache entries will be lower, as it missed a measurement, which will slightly impact the accuracy of the algorithm. To prevent the detection policy from being killed, a margin has been added to the interval which allows it to run longer if necessary, but never longer than the interval at which it is executed. The average runtime of the detection policy is 2-3 seconds under normal conditions, and has shown to reach 7-8 seconds under stress. Therefore, the final interval chosen for the detection policy is 10 seconds. For the measurement policy, measurements have shown that 2 seconds provides an optimal balance between detailed measurements and loss of data due to termination.

6.5.3 Validation

In this section, we validate the prototype implementation for the Cisco Catalyst 6500 in terms of the same requirements as for the original implementation for INVEA-TECH's FlowMon platform: (1) it should be lightweight in terms of CPU and memory utilization, (2) the accuracy should be high enough to ascertain a low number of false positives/negatives, and (3) the detection delay should be reduced to roughly 10% of conventional intrusion detection approaches. However, since the the Cisco Catalyst 6500 is a high-speed packet forwarding device that has not been designed for performing intrusion detection tasks, special care must be taken to not overload the device and possibly interrupt forwarding activities. We therefore relax the real-time requirement to detection within 30 seconds, while the CPU and memory utilization must be 10% or lower. Since we have not changed the working of the detection algorithm itself and its accuracy is invariant to the underlying implementation platform, we discuss the accuracy requirement only briefly.

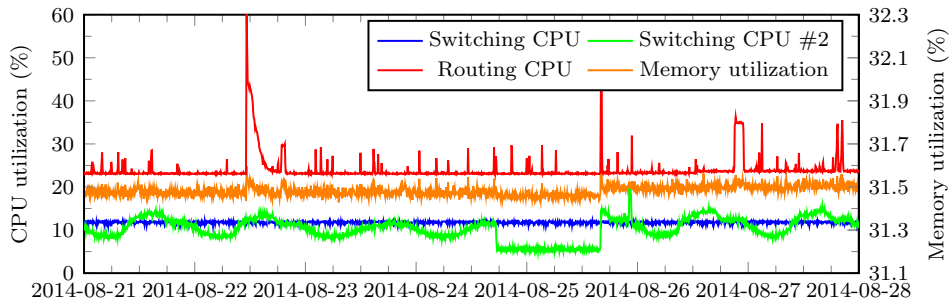


Figure 6.10: Load of the Cisco Catalyst 6500 over time.

6.5.3.1 Setup & Deployment

The prototype described in Section 6.5.2 has been developed for a *Cisco Catalyst 6500* with *Supervisor Engine 720*, running IOS 15.1(2)SY1. We have used this in combination with the *WS-X6708-10G-3C* line card for 10 Gbps Ethernet connectivity. The traffic used for validation is mirrored from the uplink of the UT campus network to the Dutch National Research and Education Network SURFnet and consists of both educational traffic, i.e., traffic generated by faculties and students, and traffic of campus residences. The link has a wire-speed of 10 Gbps with an average throughput of 1.8 Gbps during working hours. Furthermore, flow data is exported to a flow collector, such that attacks detected by the prototype can be validated manually.

The network traffic used in the original validation differs from the network traffic used here, both from its nature (backbone traffic vs. campus traffic) and volume. It is therefore clear that we have to adjust the parameters of the detection algorithm to achieve similar accuracies. As such, we have selected the optimal parameter values[2] for our observation point. For the parameter c_f, used for approximating the number of flow cache entry creations, as described in Section 6.5.1, we have measured $c_f = 59.8133$ PPF on average in our setup.

6.5.3.2 Results

The most important requirement to be validated in this work is that the implementation must be lightweight, such that the implementation does not interfere with the primary activities of the packet forwarding device, namely routing and switching. We measure the resource consumption both in terms of CPU and memory utilization. In Figure 6.10, the CPU load of the device is shown together with the memory utilization, averaged over 150 seconds. Using SNMP,

[2]The parameters used in this work are: $c_{threshold} = 4.0$, $M_{min} = 7000$, $c_{cusum} = 6.0$, $\alpha = \frac{2}{N+1}$, where $N = 540$, and $\gamma = 0.4$.

the load of the CPU is measured for three components, namely the *routing CPU*, which handles L3 traffic, and two *switching CPUs*, which process traffic at L2. Once a routing or switching decision has been made by the CPU, hardware handles subsequent packets if possible. Furthermore, the *routing CPU* also handles the network management (including the EEM), as most of this is done on L3. Consequently, our EEM policies also run on the *routing CPU*, and as such any load caused by our policies should account to the load of the *routing CPU*.

In Figure 6.10, the policies are active during the entire measurement period, even in the period from August 24 18:00 to August 25 16:00 where the switch received no data. Because the CPU utilization of most individual processes is reported as 0-1% and only peaks are reported as more than 1%, we only consider the overal CPU usage. Consequently, the overhead of managing and executing only the policies cannot be observed. This overhead is caused by processes such as the *Chunk Manager*, which handles memory allocation, *EEM Server*, which manages all EEM policies and applets, and *SNMP ENGINE*, which handles all SNMP requests. Because the overhead of operating our policies is caused by multiple processes, which also run when our implemented policies are disabled, we have measured the difference in CPU and memory utilization between operation with and without our policies. To measure this, the switch has been rebooted to clear all memory and CPU utilization. During the measurements, we have observed a load on the *routing CPU* of 4%, combined with a memory utilization of 31.3%. After enabling our policies we have observed an increase of 20% in CPU utilization, and an increase of 0.2% in memory utilization. This accounts for the average constant load added by our implementation.

During the period in which our detection algorithm was deployed, one attack passed our validation network on August 25. The attack lasted around 20 minutes and consisted of Domain Name System (DNS) reflection traffic and TCP traffic. During this attack, we only observe a minor increase of the load of the *switching CPU*, caused by the increased number of packets to be switched, and no increase in load for the *routing CPU*. As such, we conclude that the CPU load caused by our implementation during attacks does not peak and instead only consists of the constant load. The peaks in the load of the *routing CPU*, visible in Figure 6.10, are likely the effect of other routing or management processes on the Catalyst 6500, as such processes are handled by the *routing CPU*. In terms of memory utilization, we clearly observe a stable pattern in Figure 6.10. We do not observe any increase in memory utilization during the attacks, which makes us to conclude that the memory utilization does not create significant peaks.

Considering the above measurements, we conclude that the memory utilization does satisfy the requirement of using 10% of memory or less. However, the 20% CPU load caused by our implementation does not satisfy the requirement of 10% CPU utilization or less. Since the Catalyst 6500 is a packet forwarding device and not meant to perform network attack detection, such other activities should not interfere with its main purpose of operation. As a load of 20% is probable to cause interference with the routing and switching tasks, we conclude that

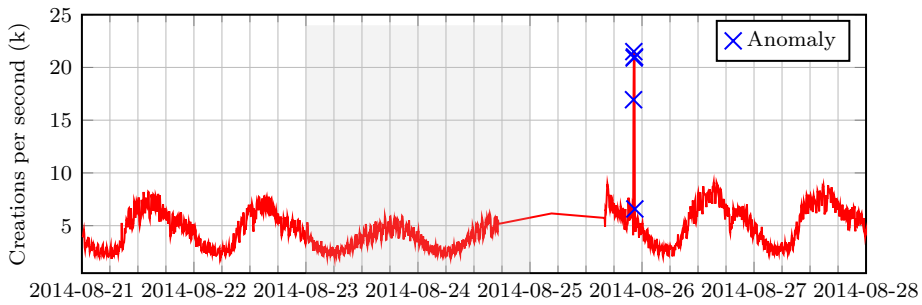


Figure 6.11: Flow cache entry creations per second (averaged per 5 minutes), as processed by the detection algorithm over time.

our implementation does not satisfy the requirement to be lightweight. The difference between the measured constant load and the lack of peaks in Figure 6.10 can be explained by the fact that the amount of traffic does not change the number of computations performed by the policies, as only the calculated values are different. Furthermore, the short and frequent execution of the policies will be averaged out to a constant added CPU load. Especially the short intervals in which the measurement policy is executed (i.e., 2 seconds), increases the load. However, increasing this interval would decrease the measurement resolution, as described in Section 6.5.1.

The second requirement is the detection delay. Our implementation uses an interval of 10 seconds between invocations of the algorithm, instead of the 5 seconds used in the original implementation, due to the runtime of the algorithm, as described in Section 6.5.2. This results in detection delays of multiples of 10 seconds, with a minimum of 10 seconds. The attack visible in Figure 6.11 was detected within the third interval, resulting in a detection delay of 30 seconds.

The final requirement considered in this work is the accuracy of the DDoS attack detection. In Figure 6.11, the number of flow cache entry creations per measurement interval is shown, averaged over 5 minute intervals. Weekends are shaded in light-gray. Diurnal patterns are clearly distinguishable and due to the nature of the traffic, we can also observe the difference between weekdays and weekends. The anomalous period around August 25 is caused by a lack of data as the switch did not receive any traffic during this period. The attack on August 25 is clearly distinguishable in Figure 6.11. It resulted in around 200% more flow records than predicted by the algorithm, and lasted for roughly 20 minutes. Multiple detection marks are shown, as the attack spanned multiple 5 minute intervals.

6.6 Conclusions

In this chapter, we have presented an extension for NetFlow and IPFIX flow exporters that detects and mitigates malicious network traffic in a near real-time fashion. Our results show that we can detect and mitigate flooding attacks within seconds, effectively preventing malicious traffic to cause flow collector overload. Additionally, we can instruct other security systems, such as firewalls, to block malicious traffic, and filter flow data of malicious traffic to prevent flow collector overload. Deployment of a prototype in the CESNET backbone network has proved both the applicability and successful operation of our approach.

Validation of our work has shown that response times as short as 5 seconds can be achieved, which easily fulfills our design target of reducing detection delays to at least 10%. This is the smallest possible value for our prototype, as it is based on measurement intervals of 5 seconds. On the one hand, related work has shown that smaller measurement intervals result in measurements with too much noise, reducing the accuracy of anomaly detection algorithms. On the other hand, larger measurement intervals result in higher detection delays and again reduced accuracy. Another requirement for our intrusion detection module has been a limited performance footprint, which has been targeted to be no more than 10%. Our prototype has shown never to consume more than 5% additional CPU time and that the additional memory usage is negligible. Finally, our prototype has a high detection rate, while maintaining a very low number of false positives. This shows that our approach is accurate.

Besides validating the prototype implementation only on a dedicated flow export platform, we have investigated whether high-end packet forwarding devices can be used for detecting, and ultimately mitigating, (DDoS) attacks in real-time. And yes, it is possible to detect DDoS attacks, which has been proven by the deployment of our prototype on a Cisco Catalyst 6500 series switch. Our results show that detection of flooding attacks is possible within tens of seconds, making real-time detection on a widely available switching platform possible. However, our prototype has also shown to cause a CPU load of 20%, which may cause interference with the routing and switching processes. According to various network operators we have stayed in touch with during this work, if the capacity of the packet forwarding device is available, it should be possible to run our DDoS attack detection in production environments. While it is possible to deploy our implementation with only 20-30% CPU capacity available, for example, it would require to be run with a lower priority, to not interfere with the routing and switching processes. As this may cause instability to our prototype, it is advised to have at least 40% CPU capacity available.

As a final note, we like to mention that the work presented in this chapter received quite some attention by industry, because of the fact that it solved an important problem for their customers: overload of the flow monitoring infrastructure due to large DDoS attacks. Even though this work has not made it into a product yet – likely because of the fact that exporter modifications are much

more difficult to maintain than regular flow analysis software – we have received word that the attention for our work is growing again; Recent high-intensity DDoS attacks are so short (seconds to minutes) that regular flow-based detection does not allow for timely detection and mitigation.

Conclusions

In this thesis, we have proposed, investigated and made a case for a new paradigm in network security monitoring, which we refer to as *compromise detection*. Compromise detection targets those incidents that actually require attention by security experts and avoids important incidents to stay under the radar of security teams due to the burden of reports that is generated every day. We therefore advocate that security analysts should focus more on *compromise* detection besides regular *attack* detection; Not just shots are important, but scored goals are.

Our approach for performing the research presented in this thesis is based on the use of flow monitoring technologies, meaning that we use network traffic aggregates instead of individual packets as input for our detection algorithms. This provides many advantages. First in terms of scalability, since it neither has stringent requirements on hardware for analyzing individual packets, nor does it require the installation of (software) agents on end systems to be able to perform detection. Second in terms of deployability, due to the fact that technologies like NetFlow and IPFIX are widely implemented on packet forwarding devices and dedicated probes, resulting in a monitoring and detection approach that allows for wide deployment.

7.1 Research Questions

In Chapter 1, we identified three research questions to guide the research presented in this work. In this section, we summarize the answers to these questions.

RQ1 – *Can flow monitoring technology be used for compromise detection?*

Over the years of working on the material presented in this thesis, we have learned that flow export is a process that is far from trivial and that, in contrast to what is often believed and advertized, flow export is anything but *plug-and-play* technology that works the same on every device. Instead, we have seen that artifacts in flow data, as well as flow export devices that do not adhere to configuration and specification, are unfortunately the rule rather than the exception (Chapter 3). Many users of flow data, such as researchers, are however not aware of this, resulting in analyses and conclusions based on data that may

be full of artifacts. Given these observations, we have evaluated whether flow data can be used for compromise detection. Our evaluation encompasses both a theoretical analysis and practical evaluation, and has resulted, among others, in a comprehensive tutorial on flow monitoring to assist researchers and operators of flow monitoring devices to perform sound measurements (Chapter 2).

For flow data to be suitable for use in compromise detection, it should ultimately reflect the original network traffic precisely. Based on the research presented in Chapter 2, we conclude that the following requirements must be satisfied:

- The configuration of the flow export device should enable the precise export of network flows. As such, packet sampling should be avoided, as it results in data loss by definition and causes our compromise detection to not be aware of the complete flows (Section 2.3). Also the configuration of expiration timers is crucial, since they determine when flows are considered to have terminated. Suboptimal configuration may cause flows to be merged into the same flow record, or flows to be split over multiple records (Section 2.4). In the case of TCP flows, as is the case of SSH and Web application compromise detection, TCP flags must be exported as well.

- When networks and devices are under attack, the resulting flow data may be an amplification of the original network packets, due to the accounting overhead imposed by the flow metering process (Section 2.6). Any overloaded component in a flow monitoring setup causes data to be lost, resulting in an unfaithful representation of the original network traffic. Measures should therefore be taken to improve the resilience of the monitoring system, which we discuss extensively in Chapter 6.

Our analysis in Chapter 3 has made clear that *verification* of the exported data is of key importance for any flow data analysis. It was shown that flow data exported by widely-spread devices from renown vendors is full of artifacts and that many devices do not adhere to configurations and specifications. This leads to interpretation errors and suboptimal functioning of detection algorithms, ultimately resulting in false positive and negative detections. Our general observation is that data from dedicated flow export devices (probes) is superior in quality to data from flow exporters that are embedded in packet forwarding devices, and that the data generally provides a precise representation of the original network traffic. We therefore conclude that the use of flow data from dedicated exporting devices is safe for use in compromise detection. Flow data from embedded devices might as well be used for compromise detection, but we have found such data to feature many artifacts, such as missing TCP flags in data from one of the most widely-deployed Cisco devices. If there is no choice for the type of flow export device and only flow data from embedded exporters is available, extensive verification (Section 3.3 and 3.4) is a necessity before the data may be used for compromise detection.

RQ2 – *How viable is compromise detection for application on the Internet?*

Once flow monitoring setups provide a means for performing sound measurements, we can use the exported flow data for the core contribution of this thesis: *compromise detection*. We target two application areas for flow-based compromise detection, namely SSH and Web applications. Our evaluations have revealed that only a minority of brute-force attacks against these targets is actually successful, i.e., resulting in a compromise. We therefore aim at providing security teams a means of identifying these compromises, such that they can focus on what is really important: quarantining compromised devices.

In Chapter 4 and 5 we have demonstrated that flow-based compromise detection is a viable detection paradigm for SSH and Web applications, respectively. For both application areas, we developed compromise detection algorithms and prototypes, and validated our work using large datasets that were collected in various networks connected to the Internet. The various validations presented throughout this thesis have shown that very high accuracies can be achieved, depending on the nature of the monitored network. This shows that our flow-based compromise detection approach is ready for production usage. Next to our own validation, we developed an open-source IDS, *SSHCure*, which was originally meant as a demonstrator for our compromise detection algorithms. Soon after releasing the first version, we have seen a major uptake of *SSHCure* by the community, resulting in large amounts of feedback on detection results. We are aware of many operational deployments of *SSHCure* in networks ranging from small Web hosting companies to large backbone operators and nation-wide CSIRTs. In addition, we have received several awards for the work on *SSHCure*, such as a Best Paper Award and a Communication Software Award.

Although outside the scope of this thesis, the viability of our compromise detection paradigm depends – besides technical correctness – on uptake and economic impact. Our intuitions and validations have shown that we can achieve a drastic reduction of attack and incident reports when security teams have to deal with compromises instead of attacks; In many situations and networks, this can result in reductions of up to 1/1000 of the total number of incident reports, as shown in Section 5.4.5 for Web applications. This reduction allows security teams to focus what is really important, i.e., compromised devices, and avoids critical incidents to stay under the radar in the mass of daily incidents.

The proper operation of flow monitoring systems and verification of the complete setup are essential for any flow data analysis, so it is for the viability of flow-based compromise detection. Moreover, due to our advanced flow data analysis used for compromise detection, the bar for flow data quality is raised even more. Some examples:

- Some of our algorithms rely on TCP flags for recognizing connection states and trigger certain analyses. However, given that many (older, hardware-based) flow exporters do not export flags for TCP connections, as shown in

Chapter 3, certain analysis can never be triggered, resulting in decreased detection performance.

- In Section 4.4, where we discuss network traffic flatness, we have shown that detection results of existing algorithms may increase significantly if the exported flow data is enhanced with fields that allow for the identification of network artifacts. Several prime examples were shown of attacks that stayed completely under the radar due to flow data deficiencies.

Our general perspective on flow data quality and tuning detection algorithms is that improving input data quality and enhancing its granularity results in a higher gain in overall detection performance than fine-tuning detection algorithms.

> **RQ3** – *Which components of flow monitoring systems are susceptible to flooding attacks, how can these attacks efficiently be detected and how can the resilience of flow monitoring systems be improved?*

In Chapter 1 and 2 we have shown that flow monitoring systems consist of various components, namely a flow exporter, flow collector and analysis application, as shown in Figure 7.1. Flow exporters use packet capturing technology that may have to be performed up to line-rates. We therefore consider them real-time devices. When under attack, flow exporter caches are filled up with attack traffic, i.e., many very small and typically short flows, effectively 'pushing out' flow cache entries for benign traffic. In those situations, modern flow exporters use special *emergency expiration* techniques in such situations to free up cache space quicker than usual by exporting cache entries prematurely. Whether full caches result in data loss very much depends on the type of flow exporter; Flow exporters with hardware caches have a hard limit on the amount of available cache space, while exporters with software caches are usually more flexible with cache management, e.g., by simply allocating more memory to the cache, if available.

Once flow exporters experience increased load under attack and start to generate more flow records, more flow export protocol messages are to be sent to

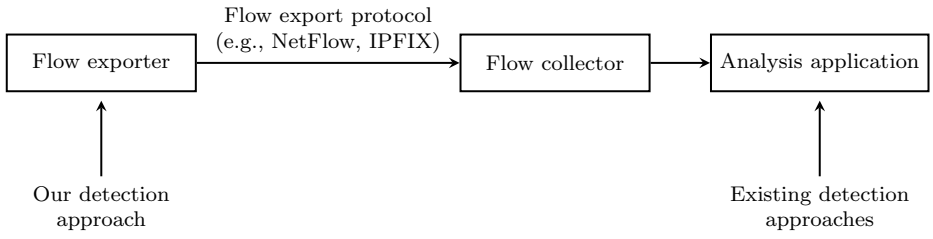


Figure 7.1: Components in a typical flow monitoring setup.

flow collectors. Special care should be taken to dimension the network between exporter and collector properly, to avoid any data loss due to dropped messages.

We initially believed that flow exporters would be most susceptible to flooding attacks, due to their real-time nature and the fact that they have to capture every individual packet for flow metering. However, contrary to that, we found that flow collectors are the most susceptible component in typical flow monitoring systems. While flow exporters are often certified to handle traffic at line-rates and come with special techniques for load management, flow collectors typically have no such countermeasures and have to do what they always do: store flow data to disk. Our interactions with vendors and operators of flow collectors have confirmed our conclusion that flow collectors are most susceptible to flooding attacks, due to the fact that they perform storage on platforms that are designed with storage capacity in mind, rather than storage performance. This may result in data loss as soon as flow collectors receive more data (over the network from flow exporters) than they can write to disk.

A straightforward approach to improve the resilience of flow monitoring systems is to increase the computational performance of the system as a whole. Given that flow collectors are generally the main bottleneck in the setup, we could use faster network interfaces between the flow exporter and collector, equip the flow collector with a faster I/O subsystem and provide the analysis application with more CPU time. We however take a different approach: In Chapter 6 we designed a lightweight anomaly detection module for flow exporters. By modelling flow cache behavior over time, we can recognize flooding attacks by imminent increases in flow cache utilization, resulting in response times of several seconds instead of minutes. We have found that dedicated flow export devices have enough spare resources for running such lightweight modules and can potentially drop attack traffic at the earliest possible occasion. Dropping attack traffic already within the flow exporter avoids overload of the network between exporter and collector, as well as congestion or data loss at flow collectors.

7.2 Discussion

Interestingly enough, industry is taking up similar ideas behind the compromise detection paradigm, albeit under a different name: *egress detection*. Egress detection targets suspicious (egress) connections towards remote Command & Control (C&C) servers, for example. Although seemingly similar to the compromise detection paradigm proposed in this thesis, the moment of detection between the two paradigms is different: While egress detection detects malicious network traffic patterns only when a compromised device is actively being misused, so *after* a compromise, our compromise detection paradigm aims at detecting a compromise *at the moment of the compromise*. This difference is crucial, as our paradigm detects compromised devices even without malware initiating (egress) network connections. Given that C&C infrastructure was found to be operated

over long periods over time [63], the moment between compromise and misuse may be long. This is an important advantage over egress detection, since compromise detection allows for quarantining compromised devices before other hosts in the same network are attacked or infected, which may not result in traffic that passes by the central observation point, for example.

The work on compromise detection presented in Chapter 4 and 5 were always meant to function as 'stand-alone' detection instances. This provided us the major advantage of being able to implement our algorithms as part of our IDS *SSHCure*, for example, and distribute to and share experiences with the public. We have however also collaborated on a more advanced, multi-layered IDS architecture that allows for chaining multiple IDSs, such that the advantages of individual IDSs could be exploited while minimizing any disadvantages. For example, a typical architecture could consist of our lightweight IDS presented in Chapter 6 as a high-speed pre-filter, which redirects any suspicious traffic to heavier-weight IDSs. In a joint work, we have proposed such an architecture, which dramatically reduces operational costs (since less expensive hardware is needed) and avoids legal barriers with respect to traffic analysis [42]. Many extensions are possible in the presented directions, such as the integration of our compromise detection algorithms in such a multi-layered architecture and the decision engine with respect to traffic forwarding that is required to make it work.

This thesis provides a solid foundation for plenty of directions for future work, of which we set out those in the remainder of this section that may have the largest impact.

Advanced applications based on flow data

Many flow-based analysis applications and appliances have been developed over the years, typically targeting rather simple applications like traffic profiling, blacklist matching, and network visualization. Although some work in the area of flow-based security monitoring exists, more advanced analysis applications, with similar requirements on flow data quality as the work in this thesis, can hardly be found. For example, our flow-based IDS *SSHCure* is one of the very few flow-based security applications, let alone open-source security applications. The reason for this may be found in the (quality of) available flow data, which may be exported suboptimally (Chapter 2), likely features flow data artifacts (Chapter 3), or does not allow the identification of network artifacts like TCP retransmission and control information (Chapter 4 and 5), which cannot be discriminated in flow data by default. We have shown in this thesis how flow measurements have to be performed, how the granularity of flow data may be improved, and how artifacts can be discovered and often overcome. Our lessons learned can be applied to *any* flow data analysis and may open doors for further research, which was not possible before.

On-probe detection modules
We have shown in Chapter 6 how to deploy software, such as lightweight detection modules, on flow exporters, in an attempt to bring analysis closer to the data source. Our evaluations have made clear that this *on-probe* approach may have a large impact on detection delays that are present in any flow monitoring systems, which may be reduced from minutes to just a few seconds. The potential of our approach is even recognized by industry, because it provides a means for monitoring and detecting specific threats that was not possible before, such as very short yet high-intensity DDoS attacks. We believe that *on-probe* detection modules have a huge potential, as it combines the advantages of flow monitoring in terms of scalability, while overcoming any disadvantages in terms of delays.

Combining host-based and network-based detection
In Chapter 1, we made clear that network-based compromise detection provides several advantages over host-based compromise detection, namely global network visibility versus visibility of a single host, and major improvements in terms of scalability. Nevertheless, due to the fact that host-based approaches usually have access to individual packets and log files, the accuracy of these approaches is somewhat higher than of their network-based counterparts. Provided that we have shown using open-source software that our network-based compromise detection is ready for production deployment, we argue that future work may investigate how the advantages of both host-based and network-based approaches may be combined, for example in a multi-layered fashion, similar to what we have proposed in [42].

Minimum Difference of Pair Assignment (MDPA)

MDPA is a distance metric for calculating how similar two histograms are. Its most distinguishing feature is that it does not satisfy the *shuffling invariance* metric, meaning that shuffling any bin values in a histogram *does* affect histogram inter-distance, as explained by means of an example in Section 5.2. In short, MDPA aims at finding the minimum difference of pair assignments between two sets, histograms in our context. As such, one has to find the best combination of one-to-one assignments such that the sum of all differences is as small as possible [5].

A.1 Calculation

The formal definition of the inter-distance of two histograms using the MDPA metric has been defined in (5.4). How the actual calculation works can best be explained based on Figure A.1, where Histogram A, B and C are taken from Figure 5.3. Distances between any two histograms are the sums of differences between pairs of samples. For example, the difference between Histogram A and B is 4. The minimum distance of pairs between Histogram A and C is 204. To illustrate the effect of MDPA not satisfying the *shuffling invariance*, we have shuffled the samples in Histogram C in Figure A.1 such that the samples are not in ascending order of their values anymore. Given that all distance permutations

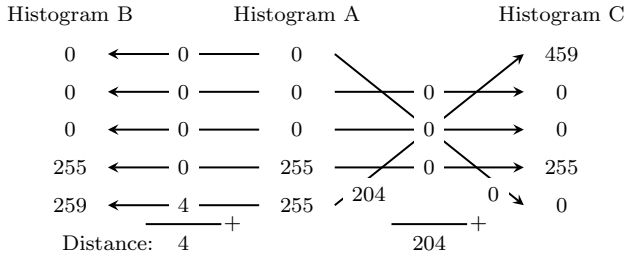


Figure A.1: MDPA calculation. Visualization based on [5].

are compared in (5.4), the overall distance between any two histograms is not affected by the order of the samples.

A.2 Normalization

Since the number of samples in a histogram is not necessarily the same between any two histograms, one can apply normalization of the histograms by multiplying all elements by a common multiple N of both histograms. One common multiple is the product of the number of samples in two histograms x and y, i.e., $n_x * n_y$. The normalized distance between two histograms is then defined as follows [5]:

$$D^N(x, y) = \frac{D(x^N, y^N)}{N} \tag{A.1}$$

Although normalization is only necessary in case of inequality of the number of samples in any set of histograms, we use the normalized distance in all our calculations for the following reasons:

- It is very unlikely that histograms within a cluster are identical in terms of the number of featured samples.
- Comparing histogram distances between different clusters can only be done if the distances are normalized.

CMS Backend URLs

The three CMSs considered for the validation of the work on Web applications presented in Chapter 5 all feature their own methods for redirecting users from a login page to their administration backend upon successful login. Table B.1 summarizes the backend login URLs, as well as CMS behaviors upon both successful and failed logins. In addition to what is listed in Table B.1, we also consider Wordpress' XML Remote Procedure Call (RPC) interface (`/xmlrpc.php`), which also requires authentication for several RPCs and has seen numerous attacks in recent times [139], [149].

CMS	**Backend login URL**	**Successful login behavior**	**Failed login behavior**
Wordpress v3.8.2	`/wp-login.php`	HTTP 302 status code, redirecting to `/wp-admin/`	HTTP 200 status code, serving login form again
Joomla v3.3.6	`/administrator/index.php`, `task=user.login`	HTTP 303 status code, redirecting to backend administration page with increased response size	HTTP 303 status code, redirecting to login form
Drupal v7.26	`/?q=user`, `/?q=user/login`, `/user/login`, `q=node&destination=node`	HTTP 302 status code, redirecting to backend administration page with increased response size	HTTP 200 status code, serving login form again

Table B.1: CMS backend login URLs and behaviors

Bibliography (refereed)

[1] S. Alcock, P. Lorier, and R. Nelson, "Libtrace: A Packet Capture and Analysis Library", *SIGCOMM Computer Communication Review*, vol. 42, no. 2, pp. 42–48, 2012.

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

[3] M. Burkhart, D. Schatzmann, B. Trammell, E. Boschi, and B. Plattner, "The Role of Network Trace Anonymization under Attack", *SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 5–11, 2010.

[4] T. Caliński and J. Harabasz, "A dendrite method for cluster analysis", *Communications in Statistics-theory and Methods*, vol. 3, no. 1, pp. 1–27, 1974.

[5] S.-H. Cha and S. N. Srihari, "On measuring the distance between histograms", *Pattern Recognition*, vol. 35, no. 6, pp. 1355–1370, 2002.

[6] K. C. Claffy, H.-W. Braun, and G. C. Polyzos, "A Parameterizable Methodology for Internet Traffic Flow Profiling", *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, pp. 1481–1494, 1995.

[7] I. Drago, R. R. Barbosa, R. Sadre, A. Pras, and J. Schönwälder, "Report of the Second Workshop on the Usage of NetFlow/IPFIX in Network Management", *Journal of Network and Systems Management*, vol. 19, no. 2, pp. 298–304, 2011.

[8] N. Duffield, "Sampling for Passive Internet Measurement: A Review", *Statistical Science*, vol. 19, no. 3, pp. 472–498, 2004.

[9] N. Duffield, C. Lund, and M. Thorup, "Estimating Flow Distributions from Sampled Flow Statistics", *IEEE/ACM Transactions on Networking*, vol. 13, no. 5, pp. 933–946, 2005.

[10] J. Fan, J. Xu, M. H. Ammar, and S. B. Moon, "Prefix-preserving IP address anonymization: measurement-based security evaluation and a new cryptography-based scheme", *Computer Networks*, vol. 46, no. 2, pp. 253–272, 2004.

[11] 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, 2014.

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

[13] W. John, S. Tafvelin, and T. Olovsson, "Passive Internet Measurement: Overview and Guidelines based on Experiences", *Computer Communications*, vol. 33, no. 5, pp. 533–550, 2010.

[14] A. Kind, M. P. Stoecklin, and X. Dimitropoulos, "Histogram-Based Traffic Anomaly Detection", *IEEE Transactions on Network and Service Management*, vol. 6, no. 2, pp. 110–121, 2009.

[15] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using OpenFlow: A Survey", *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 493–512, 2014.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks", *SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[17] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis", *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, 1987.

[18] S. Saito, K. Maruhashi, M. Takenaka, and S. Torii, "TOPASE: Detection and Prevention of Brute Force Attacks with Disciplined IPs from IDS Logs", vol. 24, no. 2, pp. 217–226, 2016.

[19] V. A. Siris and F. Papagalou, "Application of Anomaly Detection Algorithms for Detecting SYN Flooding Attacks", vol. 29, no. 9, pp. 1433–1442, 2006.

[20] 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, pp. 343–356, 2010.

[21] A. Sperotto, M. Mandjes, R. Sadre, P. T. de Boer, and A. Pras, "Autonomic Parameter Tuning of Anomaly-Based IDSs: an SSH Case Study", *IEEE Transactions on Network and Service Management*, vol. 9, no. 2, pp. 128–141, 2012.

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

[23] N. Ye, C. Borror, and Y. Zhang, "EWMA Techniques for Computer Intrusion Detection Through Anomalous Changes in Event Intensity", *Quality and Reliability Engineering International*, vol. 18, no. 6, pp. 443–451, 2002.

[24] J. L. García-Dorado, F. Mata, J. Ramos, P. M. S. del Río, V. Moreno, and J. Aracil, "High-Performance Network Traffic Processing Systems using Commodity Hardware", in *Data Traffic Monitoring and Analysis*, ser. Lecture Notes in Computer Science, vol. 7754, 2013, pp. 3–27.

[25] P. Piskac and J. Novotny, "Using of Time Characteristics in Data Flow for Traffic Classification", in *Proceedings of the 5th International Conference on Autonomous Infrastructure, Management and Security, AIMS 2011*, ser. Lecture Notes in Computer Science, vol. 6734, 2011, pp. 173–176.

[26] A. Abdou, D. Barrera, and P. C. van Oorschot, "What Lies Beneath? Analyzing Automated SSH Bruteforce Attacks", in *Proceedings of the International Conference on Passwords*, 2015.

[27] P. D. Amer and L. N. Cassel, "Management of sampled real-time network measurements", in *Proceedings of the 14th Conference on Local Computer Networks, LCN'89*, 1989, pp. 62–68.

[28] D. Bolzoni, E. Zambon, S. Etalle, and P. Hartel, "Poseidon: A 2-tier Anomaly-based Intrusion Detection System", in *Proceedings of the 4th IEEE International Information Assurance Workshop, IWIA'06*, 2006, pp. 144–156.

[29] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, "FFPF: Fairly Fast Packet Filters", in *Proceedings of the 6th Symposium on Operating Systems Design & Implementation, OSDI'04*, 2004, pp. 347–362.

[30] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, "Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware", in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC'10*, 2010, pp. 206–217.

[31] J. D. Brutlag, "Aberrant Behavior Detection in Time Series for Network Monitoring", in *Proceedings of the 14th Conference on Systems Administration, LISA 2000*, 2000, pp. 139–146.

[32] I. Cunha, F. Silveira, R. Oliveira, R. Teixeira, and C. Diot, "Uncovering Artifacts of Flow Measurement Tools", in *Proceedings of the 10th International Conference on Passive and Active Network Measurement, PAM'09*, ser. Lecture Notes in Computer Science, vol. 5448, 2009, pp. 187–196.

[33] L. Degioanni and G. Varenni, "Introducing Scalability in Network Measurement: Toward 10 Gbps with Commodity Hardware", in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, IMC'04*, 2004, pp. 233–238.

[34] M. Dell'Amico, P. Michiardi, and Y. Roudier, "Password Strength: An Empirical Analysis", in *Proceedings of IEEE INFOCOM 2010*, 2010, pp. 1–9.

[35] L. Deri, V. Lorenzetti, and S. Mortimer, "Collection and Exploration of Large Data Monitoring Sets Using Bitmap Databases", in *Traffic Monitoring and Analysis, TMA'10*, ser. Lecture Notes in Computer Science, vol. 6003, 2010, pp. 73–86.

[36] L. Deri, E. Chou, Z. Cherian, K. Karmarkar, and M. Patterson, "Increasing Data Center Network Visibility with Cisco NetFlow-Lite", in *Proceedings of the 7th International Conference on Network and Service Management, CNSM'11*, 2011, pp. 1–6.

[37] M. Drašar, "Protocol-Independent Detection of Dictionary Attacks", in *Proceedings of the 19th EUNICE/IFIP WG 6.6 International Workshop, EUNICE'13*, 2013, pp. 304–309.

[38] N. Duffield, C. Lund, and M. Thorup, "Properties and Prediction of Flow Statistics from Sampled Packet Streams", in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement, IMW'02*, 2002, pp. 159–171.

[39] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a Better NetFlow", in *Proceedings of the ACM SIGCOMM 2004 Conference, SIGCOMM'04*, 2004, pp. 245–256.

[40] F. Fusco and L. Deri, "High Speed Network Traffic Analysis with Commodity Multi-core Systems", in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC'10*, 2010, pp. 218–224.

[41] F. Fusco, M. Vlachos, and X. Dimitropoulos, "RasterZip: compressing network monitoring data with support for partial decompression", in *Proceedings of the 2012 ACM Conference on Internet Measurement, IMC'12*, 2012, pp. 51–64.

[42] M. Golling, R. Hofstede, and R. Koch, "Towards Multi-layered Intrusion Detection in High-Speed Backbone Networks", in *Proceedings of the NATO CCD COE 6th International Conference on Cyber Conflict, CyCon '14*, 2014, pp. 191–206.

[43] Y. Gu, L. Breslau, N. G. Duffield, and S. Sen, "On Passive One-Way Loss Measurements Using Sampled Flow Statistics", in *Proceedings of IEEE INFOCOM 2009*, 2009, pp. 2946–2950.

[44] L. Hellemons, L. Hendriks, R. Hofstede, A. Sperotto, R. Sadre, and A. Pras, "SSHCure: A Flow-Based SSH Intrusion Detection System", in *Dependable Networks and Services. Proceedings of the 6th International Conference on Autonomous Infrastructure, Management and Security, AIMS'12*, ser. Lecture Notes in Computer Science, vol. 7279, 2012, pp. 86–97.

[45] R. Hofstede, A. Sperotto, T. Fioreze, and A. Pras, "The Network Data Handling War: MySQL vs NfDump", in *Networked Services and Applications – Engineering, Control and Management. Proceedings of the 16th EUNICE Open European Summer School, EUNICE'10*, ser. Lecture Notes in Computer Science, vol. 6164, 2010, pp. 167–176.

[46] R. Hofstede, I. Drago, A. Sperotto, and A. Pras, "Flow Monitoring Experiences at the Ethernet-Layer", in *Energy-Aware Communications. Proceedings of the 17th EUNICE Open European Summer School, EUNICE'11*, ser. Lecture Notes in Computer Science, vol. 6955, 2011, pp. 134–145.

[47] R. Hofstede and A. Pras, "Real-Time and Resilient Intrusion Detection: A Flow-Based Approach", in *Dependable Networks and Services. Proceedings of the 6th International Conference on Autonomous Infrastructure, Management and Security, AIMS 2012*, 2012, pp. 109–112.

[48] R. Hofstede, V. Bartoš, A. Sperotto, and A. Pras, "Towards Real-Time Intrusion Detection for NetFlow/IPFIX", in *Proceedings of the 9th International Conference on Network and Service Management, CNSM'13*, 2013, pp. 227–234.

[49] R. Hofstede, I. Drago, A. Sperotto, R. Sadre, and A. Pras, "Measurement Artifacts in NetFlow Data", in *Proceedings of the 14th International Conference on Passive and Active Measurement, PAM'13*, ser. Lecture Notes in Computer Science, vol. 7799, 2013, pp. 1–10.

[50] R. Hofstede, M. Jonker, A. Sperotto, and A. Pras, "Web Application Brute-force Attack & Compromise Detection using IPFIX", (under review), 2016.

[51] M. Javed and V. Paxson, "Detecting Stealthy, Distributed SSH Brute-Forcing", in *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, 2013.

[52] M. Jonker, R. Hofstede, A. Sperotto, and A. Pras, "Unveiling Flat Traffic on the Internet: An SSH Attack Case Study", in *Proceedings of the 14th IFIP/IEEE Symposium on Integrated Network and Service Management, IM'15*, 2015, pp. 270–278.

[53] J. Kögel, "One-way Delay Measurement based on Flow Data: Quantification and Compensation of Errors by Exporter Profiling", in *Proceedings of the International Conference on Information Networking, ICOIN'11*, 2011, pp. 25–30.

[54] R. R. Kompella and C. Estan, "The Power of Slicing in Internet Flow Measurement", in *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement, IMC'05*, 2005, pp. 105–118.

[55] F. B. Manolache, Q. Hou, and O. Rusu, "Analysis and Prevention of Network Password Guessing Attacks in an Enterprise Environment", in *Networking in Education and Research, Joint Event 13th RoEduNet & 8th RENAM Conference*, 2014, pp. 1–7.

[56] J. Micheel, S. Donnelly, and I. Graham, "Precision Timestamping of Network Packets", in *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement, IMW'01*, 2001, pp. 273–277.

[57] C. Morariu, P. Racz, and B. Stiller, "SCRIPT: A Framework for Scalable Real-time IP Flow Record Analysis", in *Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium, NOMS'10*, 2010, pp. 278–285.

[58] G. Münz and G. Carle, "Application of Forecasting Techniques and Control Charts for Traffic Anomaly Detection", in *Proceedings of the 19th ITC Specialist Seminar on Network Usage and Traffic*, 2008.

[59] A.-C. Orgerie, P. Gonçalves, M. Imbert, J. Ridoux, and D. Veitch, "Survey of Network Metrology Platforms", in *2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet, SAINT'12*, 2012, pp. 220–225.

[60] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending Networking into the Virtualization Layer", in *8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[61] A. Pras, J. J. Santanna, J. Steinberger, and A. Sperotto, "DDoS 3.0 – How terrorists bring down the Internet", in *Proceedings of 18th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems and Dependability and Fault-Tolerance (MMB & DFT)*, 2016.

[62] H. Qiu, N. Eklund, X. Hu, W. Yan, and N. Iyer, "Anomaly Detection using Data Clustering and Neural Networks", in *Proceedings of the IEEE International Joint Conference on Neural Networks, 2008, IJCNN'08*, 2008, pp. 3627–3633.

[63] C. Rossow, C. Dietrich, and H. Bos, "Large-Scale Analysis of Malware Downloaders", in *Proceedings of the 12th International Conference on the Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA'12*, 2012, pp. 42–61.

[64] R. Sadre, A. Sperotto, and A. Pras, "The Effects of DDoS Attacks on Flow Monitoring Applications", in *Proceedings of the 13th IEEE/IFIP Network Operations and Management Symposium, NOMS'12*, 2012, pp. 269–277.

[65] 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 *Proceedings of the 14th IFIP/IEEE Symposium on Integrated Network and Service Management, IM'15*, 2015, pp. 243–251.

[66] S. Seetharaman, "OpenFlow/SDN tutorial OFC/NFOEC", in *Optical Fiber Communication Conference and Exposition (OFC/NFOEC), 2012 and the National Fiber Optic Engineers Conference*, 2012, pp. 1–52.

[67] R. Sommer and A. Feldmann, "NetFlow: Information loss or win?", in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement, IMW'02*, 2002, pp. 173–174.

[68] A. Sperotto, R. Sadre, P.-T. de Boer, and A. Pras, "Hidden Markov Model modeling of SSH brute-force attacks", in *Integrated Management of Systems, Services, Processes and People in IT. Proceedings of the 20th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM'09*, ser. Lecture Notes in Computer Science, vol. 5841, 2009, pp. 164–176.

[69] D. van der Steeg, R. Hofstede, A. Sperotto, and A. Pras, "Real-time DDoS attack detection for Cisco IOS using NetFlow", in *Proceedings of the 14th IFIP/IEEE Symposium on Integrated Network and Service Management, IM'15*, 2015, pp. 972–977.

[70] J. Steinberger, L. Schehlmann, S. Abt, and H. Baier, "Anomaly Detection and Mitigation at Internet scale: A Survey", in *Proceedings of the 7th International Conference on Autonomous Infrastructure, Management and Security, AIMS'13*, ser. Lecture Notes in Computer Science, vol. 7943, 2013, pp. 49–60.

[71] O. van der Toorn, R. Hofstede, M. Jonker, and A. Sperotto, "A First Look at HTTP(S) Intrusion Detection using NetFlow/IPFIX", in *Proceedings of the 14th IFIP/IEEE Symposium on Integrated Network and Service Management, IM'15*, 2015, pp. 862–865.

[72] B. Trammell, B. Tellenbach, D. Schatzmann, and M. Burkhart, "Peeling away Timing Error in NetFlow Data", in *Proceedings of the 12th International Conference on Passive and Active Measurement, PAM'11*, ser. Lecture Notes in Computer Science, vol. 6579, 2011, pp. 194–203.

[73] P. Velan, "Practical Experience with IPFIX Flow Collectors", in *Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management, IM'13*, 2013, pp. 1015–1020.

[74] M. Vizváry and J. Vykopal, "Flow-based detection of RDP brute-force attacks", in *Proceedings of 7th International Conference on Security and Protection of Information, SPI'13*, 2013, pp. 131–137.

[75] J. Vykopal, T. Plesnik, and P. Minarik, "Network-Based Dictionary Attack Detection", in *Proceedings of the 2009 International Conference on Future Networks*, 2009, pp. 23–27.

[76] H. Wang, D. Zhang, and K. Shin, "Detecting SYN flooding attacks", in *Proceedings of IEEE INFOCOM 2002*, vol. 3, 2002, pp. 1530–1539.

[77] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "FlowSense: Monitoring Network Utilization with Zero Measurement Cost", in *Proceedings of the 14th International Conference on Passive and Active Measurement, PAM'13*, ser. Lecture Notes in Computer Science, vol. 7799, 2013, pp. 31–41.

[78] J. Zhang and A. Moore, "Traffic trace artifacts due to monitoring via port mirroring", in *Proceedings of the 15th IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services, E2EMON'07*, 2007, pp. 1–8.

[79] M. Drašar, "Behavioral Detection of Distributed Dictionary Attacks", PhD thesis, Masaryk University, Brno, Czech Republic, 2015.

[80] R. H. Koch, "Systemarchitektur zur Ein- und Ausbruchserkennung in verschlüsselten Umgebungen", PhD thesis, Universität der Bundeswehr München, München, Germany, 2015.

[81] A. Sperotto, "Flow-Based Intrusion Detection", PhD thesis, University of Twente, Enschede, The Netherlands, 2010.

[82] M. J. Quinn, "Analysis of Real-World Passwords for Social Media Sites", PhD thesis, James Madison University, Harrisonburg, Virginia (United States), 2015.

[83] J. Vykopal, "Flow-based Brute-force Attack Detection in Large and High-speed Networks", PhD thesis, Masaryk University, Brno, Czech Republic, 2013.

Bibliography

[84] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 2009, vol. 344.

[85] M. W. Lucas, *Absolute OpenBSD: Unix for the Practical Paranoid*. No Starch Press, 2003.

[86] A. Orebaugh, G. Ramirez, J. Burke, L. Pesce, J. Wright, and G. Morris, *Chapter 6 – Wireless Sniffing with Wireshark*. Syngress, 2007, pp. 267–370.

[87] R. Braden, *Requirements for Internet Hosts – Communication Layers*, RFC 1122 (Internet Standard), Internet Engineering Task Force, 1989. [Online]. Available: `http://www.ietf.org/rfc/rfc1122.txt`.

[88] C. Mills, D. Hirsh, and G. Ruth, *Internet Accounting: Background*, RFC 1272, Internet Engineering Task Force, 1991. [Online]. Available: `http://www.ietf.org/rfc/rfc1272.txt`.

[89] N. Brownlee, *RTFM: Applicability Statement*, RFC 2721 (Informational), Internet Engineering Task Force, 1999. [Online]. Available: `http://www.ietf.org/rfc/rfc2721.txt`.

[90] R. R. Stewart, M. A. Ramalho, Q. Xie, M. Tuexen, and P. T. Conrad, *Stream Control Transmission Protocol (SCTP) Partial Reliability Extension*, RFC 3758 (Standards Track), Internet Engineering Task Force, 2004. [Online]. Available: `http://www.ietf.org/rfc/rfc3758.txt`.

[91] J. Quittek, T. Zseby, B. Claise, and S. Zander, *Requirements for IP Flow Information Export*, RFC 3917 (Informational), Internet Engineering Task Force, 2004. [Online]. Available: `http://www.ietf.org/rfc/rfc3917.txt`.

[92] B. Claise, *Cisco Systems NetFlow Services Export Version 9*, RFC 3954 (Informational), Internet Engineering Task Force, 2004. [Online]. Available: `http://www.ietf.org/rfc/rfc3954.txt`.

[93] S. Leinen, *Evaluation of Candidate Protocols for IP Flow Information Export*, RFC 3955 (Informational), Internet Engineering Task Force, 2004. [Online]. Available: `http://www.ietf.org/rfc/rfc3955.txt`.

[94] R. R. Stewart, *Stream Control Transmission Protocol*, RFC 4960 (Standards Track), Internet Engineering Task Force, 2007. [Online]. Available: `http://www.ietf.org/rfc/rfc4960.txt`.

[95] B. Trammell and E. Boschi, *Bidirectional Flow Export Using IP Flow Information Export (IPFIX)*, RFC 5103 (Standards Track), Internet Engineering Task Force, 2008. [Online]. Available: `http://www.ietf.org/rfc/rfc5103.txt`.

[96] E. Boschi, L. Mark, J. Quittek, M. Stiemerling, and P. Aitken, *IP Flow Information Export (IPFIX) Implementation Guidelines*, RFC 5153 (Informational), Internet Engineering Task Force, 2008. [Online]. Available: `http://www.ietf.org/rfc/rfc5153.txt`.

[97] G. Sadasivan, N. Brownlee, B. Claise, and J. Quittek, *Architecture for IP Flow Information Export*, RFC 5470 (Informational), Internet Engineering Task Force, 2009. [Online]. Available: `http://www.ietf.org/rfc/rfc5470.txt`.

[98] T. Zseby, E. Boschi, N. Brownlee, and B. Claise, *IP Flow Information Export (IPFIX) Applicability*, RFC 5472 (Internet Standard), Internet Engineering Task Force, 2009. [Online]. Available: `http://www.ietf.org/rfc/rfc5472.txt`.

[99] T. Zseby, M. Molina, N. Duffield, S. Niccolini, and F. Raspall, *Sampling and Filtering Techniques for IP Packet Selection*, RFC 5475 (Proposed Standard), Internet Engineering Task Force, 2009. [Online]. Available: `http://www.ietf.org/rfc/rfc5475.txt`.

[100] B. Claise, A. Johnson, and J. Quittek, *Packet Sampling (PSAMP) Protocol Specifications*, RFC 5476 (Proposed Standard), Internet Engineering Task Force, 2009. [Online]. Available: `http://www.ietf.org/rfc/rfc5476.txt`.

[101] B. Trammell, E. Boschi, L. Mark, T. Zseby, and A. Wagner, *Specification of the IP Flow Information Export File Format*, RFC 5655 (Proposed Standard), Internet Engineering Task Force, 2009. [Online]. Available: `http://www.ietf.org/rfc/rfc5655.txt`.

[102] B. Claise, G. Dhandapani, P. Aitken, and S. Yates, *Export of Structured Data in IP Flow Information Export (IPFIX)*, RFC 6313 (Standards Track), Internet Engineering Task Force, 2011. [Online]. Available: `http://www.ietf.org/rfc/rfc6313.txt`.

[103] B. Claise, P. Aitken, A. Johnson, and G. Münz, *IP Flow Information Export (IPFIX) Per Stream Control Transmission Protocol (SCTP) Stream*, RFC 6526 (Standards Track), Internet Engineering Task Force, 2012. [Online]. Available: `http://www.ietf.org/rfc/rfc6526.txt`.

[104] B. Claise, B. Trammell, and P. Aitken, *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*, RFC 7011 (Internet Standard), Internet Engineering Task Force, 2013. [Online]. Available: `http://www.ietf.org/rfc/rfc7011.txt`.

[105] B. Claise and B. Trammell, *Information Model for IP Flow Information Export*, RFC 7012 (Standards Track), Internet Engineering Task Force, 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7012.txt.

[106] B. Trammell and B. Claise, *Guidelines for Authors and Reviewers of IP Flow Information Export (IPFIX) Information Elements*, RFC 7013 (Best Current Practice), Internet Engineering Task Force, 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7013.txt.

[107] S. D'Antonio, T. Zseby, C. Henke, and L. Peluso, *Flow Selection Techniques*, RFC 7014 (Standards Track), Internet Engineering Task Force, 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7014.txt.

[108] B. Trammell, A. Wagner, and B. Claise, *Flow Aggregation for the IP Flow Information Export Protocol*, RFC 7015 (Standards Track), Internet Engineering Task Force, 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7015.txt.

[109] P. Aitken, B. Claise, S. B.S., C. McDowall, and J. Schönwälder, *Exporting MIB Variables using the IPFIX Protocol*, Internet Engineering Task Force, 2014. [Online]. Available: http://www.ietf.org/id/draft-ietf-ipfix-mib-variable-export-05.txt.

[110] M. Burnett, *Yes, 123456 is the most common password, but here's why that's misleading*, Accessed on 22 May 2016, 2015. [Online]. Available: http://arstechnica.com/security/2015/01/yes-123456-is-the-most-common-password-but-heres-why-thats-misleading/.

[111] Ars Technica, *Attackers trick 162,000 WordPress sites into launching DDoS attack*, Accessed on 22 May 2016, 2014. [Online]. Available: http://arstechnica.com/security/2014/03/more-than-162000-legit-wordpress-sites-abused-in-powerful-ddos-attack/.

[112] Best Host News, *cPanel vs. Plesk vs. DirectAdmin comparison*, Accessed on 22 May 2016, 2015. [Online]. Available: https://www.besthostnews.com/cpanel-vs-plesk-vs-directadmin/.

[113] Cooperative Association for Internet Data Analysis (CAIDA), *Summary of Anonymization Best Practice Techniques*, Accessed on 22 May 2016, 2010. [Online]. Available: http://www.caida.org/projects/predict/anonymization/.

[114] Cisco Systems, *Cisco Catalyst 6500 Architecture White Paper*, Accessed on 22 May 2016, 2013. [Online]. Available: http://www.cisco.com/en/US/prod/collateral/switches/ps5718/ps708/prod_white_paper0900aecd80673385.html.

[115] ——, *Catalyst 6500 Series Switch Cisco IOS Software Configuration Guide*, Accessed on 22 May 2016, 2009. [Online]. Available: http://www.cisco.com/en/US/docs/switches/lan/catalyst6500/ios/12.2SXF/native/configuration/guide/122sxscg.pdf.

[116] ——, *Cisco IOS Flexible NetFlow*, Accessed on 22 May 2016, 2008. [Online]. Available: http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps6601/ps6965/product_data_sheet0900aecd804b590b.html.

[117] ——, *Cisco IOS Flexible NetFlow Commands*, Accessed on 22 May 2016, 2010. [Online]. Available: http://www.cisco.com/en/US/docs/ios/fnetflow/command/reference/fnf_01.pdf.

[118] ——, *Catalyst 6500/6000 Switch High CPU Utilization*, Accessed on 22 May 2016, 2012. [Online]. Available: http://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/63992-6k-high-cpu.html.

[119] ——, *Cisco IOS NetFlow and Security*, Accessed on 22 May 2016, 2005. [Online]. Available: http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6586/ps6642/prod_presentation0900aecd80311f49.pdf.

[120] ——, *NetFlow Services Solutions Guide*, Accessed on 22 May 2016, 2007. [Online]. Available: http://www.cisco.com/en/US/docs/ios/solutions_docs/netflow/nfwhite.html.

[121] ——, *Switched Port Analyzer (SPAN)*, Accessed on 22 May 2016, 2007. [Online]. Available: http://www.cisco.com/en/US/tech/tk389/tk816/tk834/tsd_technology_support_sub-protocol_home.html.

[122] Cloudmark, *Spammers hacking web servers to host porn, send spam*, Accessed on 22 May 2016, 2013. [Online]. Available: http://blog.cloudmark.com/2013/04/22/spammers-hacking-web-pages-to-host-porn/.

[123] J. H. Follett, *Cisco: Catalyst 6500 The Most Successful Switch Ever*, Accessed on 22 May 2016, 2006. [Online]. Available: http://www.crn.com/news/networking/189500982/cisco-catalyst-6500-the-most-successful-switch-ever.htm.

[124] T. Wilson, *Poor Priorities, Lack Of Resources Put Enterprises At Risk, Security Pros Say*, Accessed on 22 May 2016, 2015. [Online]. Available: http://www.darkreading.com/vulnerabilities---threats/poor-priorities-lack-of-resources-put-enterprises-at-risk-security-pros-say/d/d-id/1321308.

[125] European Parliament & Council, *Directive 2006/24/EC of the European Parliament and of the Council of 15 March 2006 on the retention of data generated or processed in connection with the provision of publicly available electronic communications services or of public communications networks and amending Directive 2002/58/EC*, Accessed on 22 May 2016, 2006. [Online]. Available: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2006:105:0054:0063:EN:PDF.

[126] Hunton & Williams, *New Dutch Law Introduces General Data Breach Notification Obligation and Higher Sanctions*, Accessed on 22 May 2016, 2015. [Online]. Available: https://www.huntonprivacyblog.com/2015/06/02/new-dutch-law-introduces-general-data-breach-notification-obligation-higher-sanctions/.

[127] IANA, *IP Flow Information Export (IPFIX) Entities*, Accessed on 22 May 2016, 2013. [Online]. Available: http://www.iana.org/assignments/ipfix/ipfix.xml.

[128] B. Trammell and B. Claise, *Applying IPFIX to Network Measurement and Management*, Accessed on 22 May 2016, 2013. [Online]. Available: http://www.ietf.org/edu/tutorials/ipfix-tutorial.pdf.

[129] IETF, *IP Flow Information Export (ipfix)*, Accessed on 22 May 2016. [Online]. Available: http://datatracker.ietf.org/wg/ipfix/charter/.

[130] Incapsula, *Why CMS Platforms Are Common Hacking Targets (and what to do about it)*, Accessed on 22 May 2016, 2014. [Online]. Available: https://www.incapsula.com/blog/cms-security-tips.html.

[131] Juniper Networks, *flow-inactive-timeout – Technical Documentation*, Accessed on 22 May 2016, 2012. [Online]. Available: http://www.juniper.net/techpubs/en_US/junos/topics/reference/configuration-statement/flow-inactive-timeout-edit-forwarding-options.html.

[132] KrebsOnSecurity, *Brute Force Attacks Build WordPress Botnet*, Accessed on 22 May 2016, 2013. [Online]. Available: http://krebsonsecurity.com/2013/04/brute-force-attacks-build-wordpress-botnet/.

[133] OpenBL.org, *Statistics*, Accessed on 22 May 2016, 2015. [Online]. Available: http://www.openbl.org/statistics.html.

[134] Open Networking Foundation, *OpenFlow Switch Specification*, Accessed on 22 May 2016, 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf.

[135] OpenSSH.com, *Implementations on the Internet over time*, Accessed on 22 May 2016, 2008. [Online]. Available: http://www.openssh.com/usage/graphs.html.

[136] M. Patterson, *NetFlow v9 vs. NetFlow v5: What Are the Differences?*, Accessed on 22 May 2016, 2009. [Online]. Available: http://www.plixer.com/blog/netflow/netflow-v9-vs-netflow-v5/.

[137] C. Schmoll, S. Teofili, E. Delzeri, G. Bianchi, I. Gojmerac, E. Hyytia, B. Trammell, E. Boschi, G. Lioudakis, F. Gogoulos, A. Antonakopoulou, D. Kaklamani, and I. Venieris, *State of the art on data protection algorithms for monitoring systems: FP7-PRISM IST-2007-215350 Deliverable 3.1.1*, Accessed on 22 May 2016, 2008. [Online]. Available: http://fp7-prism.eu/images/upload/Deliverables/fp7-prism-wp3.1-d3.1.1-final.pdf.

[138] QoSient, *Argus, the network Audit Record Generation and Utilization System*, Accessed on 22 May 2016, 2015. [Online]. Available: http://qosient.com/argus/.

[139] J. Reilink, *Huge increase in WordPress xmlrpc.php POST requests*, Accessed on 22 May 2016, 2014. [Online]. Available: https://www.saotn.org/huge-increase-wordpress-xmlrpc-php-post-requests/.

[140] SC Magazine, *Four teens, adult arrested for Ziggo DDoS attacks*, Accessed on 22 May 2016, 2015. [Online]. Available: http://www.scmagazine.com/four-teens-adult-arrested-for-attacks-against-netherlands-internet-provider/article/443675/.

[141] SEC Consult, *The Omnipresence of Ubiquiti Networks Devices on the Public Web*, Accessed on 22 May 2016, 2015. [Online]. Available: http://blog.sec-consult.com/2015/11/the-omnipresence-of-ubiquiti-networks.html.

[142] P. Phaal, *sFlow Version 5*, Accessed on 22 May 2016, 2004. [Online]. Available: http://sflow.org/sflow_version_5.txt.

[143] Spamhaus, *Spam through compromised passwords: can it be stopped?*, Accessed on 22 May 2016, 2012. [Online]. Available: http://www.spamhaus.org/news/article/681/spam-through-compromised-passwords-can-it-be-stopped.

[144] Sucuri, *Big Increase in Distributed Brute Force Attacks Against Joomla Websites*, Accessed on 22 May 2016, 2013. [Online]. Available: https://blog.sucuri.net/2013/09/big-increase-in-distributed-brute-force-attacks-against-joomla-websites.html.

[145] ——, *SSH Brute Force - The 10 Year Old Attack That Still Persists*, Accessed on 22 May 2016, 2015. [Online]. Available: https://blog.sucuri.net/2013/07/ssh-brute-force-the-10-year-old-attack-that-still-persists.html.

[146] ——, *WordPress Brute Force Attacks 2015 Threat Landscape*, Accessed on 22 May 2016, 2015. [Online]. Available: https://blog.sucuri.net/2015/09/wordpress-brute-force-attacks-2015-threat-landscape.html.

[147] ——, *WordPress Brute Force Attacks*, Accessed on 22 May 2016, 2015. [Online]. Available: https://sucuri.net/security-reports/brute-force/.

[148] ——, *WordPress Malware Active VisitorTracker Campaign*, Accessed on 22 May 2016, 2015. [Online]. Available: https://blog.sucuri.net/2015/09/wordpress-malware-active-visitortracker-campaign.html.

[149] D. Cid, *Brute Force Amplification Attacks Against WordPress XMLRPC*, Accessed on 22 May 2016, 2015. [Online]. Available: https://blog.sucuri.net/2015/10/brute-force-amplification-attacks-against-wordpress-xmlrpc.html.

[150] The Hacker News, *WordPress Security: Brute Force Amplification Attack Targeting Thousand of Blogs*, Accessed on 22 May 2016, 2015. [Online]. Available: http://thehackernews.com/2015/10/WordPress-BruteForce-Amplification.html.

[151] Threatpost, *WordPress Sites Seeing Increased Malware, Brute Force Attacks This Week*, Accessed on 22 May 2016, 2004. [Online]. Available: https://threatpost.com/wordpress-sites-seeing-increased-malware-brute-force-attacks-this-week/107378/.

[152] The Next Web, *Check your security settings: Brute force attacks against WordPress and Joomla sites have tripled*, Accessed on 22 May 2016, 2013. [Online]. Available: http://thenextweb.com/insider/2013/04/12/wordpress-and-joomla-sites-see-massive-increase-in-brute-force-attacks/.

[153] W3Techs, *Historical yearly trends in the usage of content management systems for websites*, Accessed on 22 May 2016. [Online]. Available: http://w3techs.com/technologies/history_overview/content_management/all/y.

[154] J. Weber, *The Fundamentals of Passive Monitoring Access*, Accessed on 22 May 2016, 2006. [Online]. Available: http://www.nanog.org/meetings/nanog37/presentations/joy-weber.pdf.

[155] Cisco Systems, "Introduction to Cisco IOS NetFlow – A Technical Overview", White paper, 2012.

[156] Imperva, "2015 Web Application Attack Report (WAAR)", Tech. Rep., 2015.

[157] Level 3 Communications, "Safeguarding the Internet", Tech. Rep., 2015. [Online]. Available: http://www.level3.com/~/media/files/white-paper/en_secur_wp_botnetresearchreport.ashx.

[158] McAfee Labs, "Threats Report - May 2015", Tech. Rep., 2015. [Online]. Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2015.pdf.

[159] L. Metcalf and J. M. Spring, "Everything you wanted to know about blacklists but were afraid to ask (CERTCC-2013-39)", Tech. Rep., 2013.

[160] Venafi, "Ponemon 2014 SSH Security Vulnerability Report", Tech. Rep., 2014.

[161] Radware, "Global Application & Network Security Report 2014-2015", White paper, 2014.

Acronyms

ACL Access Control List
AJAX Asynchronous JavaScript And XML
APT Advanced Persistent Threat
ARP Address Resolution Protocol
ASIC Application Specific Integrated Circuits

BPF Bytes per Flow
BYOD Bring Your Own Device

C&C Command & Control
CAM Content-Addressable Memory
CDF Cumulative Distribution Function
CH Calinski & Harabasz
CLI Command-Line Interface
CMS Content Management System
CPU Central Processing Unit
CSIRT Computer Security Incident Response Team
CUSUM CUmulative SUM

DBMS Database Management System
DDoS Distributed Denial of Service
DNS Domain Name System
DNSSEC Domain Name System Security Extensions
DoS Denial of Service
DPI Deep Packet Inspection
DR Detection Rate

EWMA Exponentially Weighted Moving Average

FN False Negative
FNR False Negative Rate
FP False Positive
FPR False Positive Rate

HCA Hierarchical Cluster Analysis
HMM Hidden Markov Model
HTML HyperText Markup Language
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure

I/O Input/Output
IANA Internet Assigned Numbers Authority
ICAM Internal Content-Addressable Memory
IDS Intrusion Detection System
IE Information Element
IETF Internet Engineering Task Force
IOT Internet of Things
IP Internet Protocol
IPFIX IP Flow Information Export
ISP Internet Service Provider

LAN Local Area Network

MAC Media Access Control
MDPA Minimum Difference of Pair Assignments
MIB Management Information Base
MPLS MultiProtocol Label Switching
MSS Maximum Segment Size
MTU Maximum Transmission Unit

NAT Network Address Translation
NREN National Research and Education Network

PHP PHP: Hypertext Preprocessor
PPF Packets per flow
PSAMP Packet SAMPling

RDP Remote Desktop Protocol
RFC Request for Comment
RFI Remote File Inclusion
ROC Receiver Operating Characteristics
RPC Remote Procedure Call
RTFM Real-time Traffic Flow Measurement

RTO Retransmission TimeOut
RTT Round Trip Time

SCTP Stream Control Transmission Protocol
SDN Software-Defined Networking
SNMP Simple Network Management Protocol
SRTT Smoothed Round-Trip Time
SSH Secure SHell
SSL Secure Sockets Layer

TCAM Ternary Content-Addressable Memory
TCL Tool Command Language
TCP Transmission Control Protocol
TLS Transport Layer Security
TN True Negative
TNR True Negative Rate
ToS Type of Service
TP True Positive
TPR True Positive Rate

UDP User Datagram Protocol
URL Uniform Resource Locator
UT University of Twente

VLAN Virtual Local Area Network
VNC Virtual Network Computing

WG Working Group

About the Author

Rick Hofstede was born in Ulm, Germany, in 1988 and moved with his family to the Achterhoek in the Netherlands at the age of ten. After completing his pre-university education at Het Assink Lyceum in Haaksbergen in 2006, he started his B.Sc. studies in Telematics at the University of Twente. Already during his B.Sc. programme, Rick worked with Ph.D. students at the Design & Analysis of Communication Systems (DACS) group, where he published his first paper at the IFIP/IEEE International Symposium on Integrated Network Management, IM 2009, in New York. After completing his B.Sc. studies, Rick continued his Telematics studies at the University of Twente and graduated *cum laude* in 2009. A dedicated M.Sc. programme was designed that allowed for continuing the work within the DACS group and to foster closer collaborations with the group's Ph.D. students.

After his graduation, Rick decided to follow-up on the work that was started during the M.Sc. programme, but now as a Ph.D. student. First within the context of the EU FP7 project UniverSelf, later also within the EU FP7 project SALUS, the EU FP7 Network of Excellence (NoE) project FLAMINGO and several other smaller projects, both national and international. The FLAMINGO project provided Ph.D. students a means to obtain a joint Ph.D. degree with foreign universities and FLAMINGO partners. In Rick's particular case, a close collaboration with the Universität der Bundeswehr München, Germany, was established and a joint Ph.D. degree was realized. During the last year of his Ph.D. project, Rick started working for the Dutch start-up company RedSocks, where he became responsible for the technical development of the company's networking appliances, which feature some of the ideas and findings presented in this thesis. The RedSocks appliances are fully related to exporting, collecting and analyzing flow data, and working for RedSocks therefore seemed an obvious choice that allowed Rick to explore the industrial side of the field.

In the context of open-source software, Rick is known from two projects in particular. First, SURFmap,[1] a network monitoring tool based on the Google Maps API, which became a widely used monitoring tool for visualizing flow data in a Web browser. SURFmap was the topic of Rick's first paper in 2009 and remained under active development until 2015. Second, SSHCure,[2] the first flow-based IDS that could identify SSH compromises in flow data and one of the key contributions of this thesis. Both open-source projects have seen a major uptake by both academia and industry.

[1] `http://surfmap.sf.net/`

[2] `http://sshcure.sf.net/`, followed up (in 2015) by `https://github.com/sshcure/sshcure/`

Publications by the Author

This is a comprehensive list of publications by the author, sorted in reverse chronological order.

- R. Hofstede, M. Jonker, A. Sperotto, A. Pras. *Flow-based Web Application Brute-force Attack & Compromise Detection* (under review)
- R. Hofstede, A. Pras, A. Sperotto, G. Dreo Rodosek. *From Intrusion Detection to Compromise Detection: A Flow-based Approach* (under review)
- O. Festor, A. Lahmadi, R. Hofstede, A. Pras. *Extending IP Flow-Based Network Monitoring with Location Information* (under review – intended IETF RFC status: informational)
- I. Drago, R. Hofstede, R. Sadre, A. Sperotto, A. Pras. *Measuring Cloud Service Health Using NetFlow/IPFIX: The WikiLeaks Case*. In: Journal of Network and Systems Management (JNSM), 2015, pp. 58-88.
- R. Hofstede, L. Hendriks. *Unveiling SSHCure 3.0: Flow-based SSH Compromise Detection*. In: Proceedings of the International Conference on Networked Systems, NetSys 2015, 9-13 March 2015, Cottbus, Germany – **Communication Software Award**
- R. Hofstede, L. Hendriks. *SSH Compromise Detection Using NetFlow/IPFIX*. In: FloCon 2015, 12-15 January 2015, Portland, Oregon, USA.
- A. Sperotto, R. Hofstede, A. Dainotti, C. Schmitt, G. Dreo Rodosek. *Special Issue of the International Journal on Network Management (IJNM) on "Measure, Detect and Mitigate – Challenges and Trends in Network Security" – Editorial*. In: International Journal on Network Management (IJNM), 2014, pp. 261-262.
- J.J. Cardoso de Santanna, R.M. van Rijswijk-Deij, R. Hofstede, A. Sperotto, M. Wierbosch, L. Zambenedetti Granville, A. Pras. *Booters – An Analysis of DDoS-as-a-Service Attacks*. In: Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015, 11-15 May 2015, Ottawa, Canada. pp. 243-251.
- M. Jonker, R. Hofstede, A. Sperotto, A. Pras. *Unveiling Flat Traffic on the Internet: An SSH Attack Case Study*. In: Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015, 11-15 May 2015, Ottawa, Canada. pp. 270-278.
- O. van der Toorn, R. Hofstede, M. Jonker, A. Sperotto. *A First Look at HTTP(S) Intrusion Detection using NetFlow/IPFIX*. In: Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015, 11-15 May 2015, Ottawa, Canada. pp. 862-865.
- D. van der Steeg, R. Hofstede, A. Sperotto, A. Pras. *Real-time DDoS attack detection for Cisco IOS using NetFlow*. In: Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015, 11-15 May 2015, Ottawa, Canada. pp. 972-977.
- M. Golling, R. Koch, P. Hillmann, R. Hofstede, F. Tietze. *YANG2UML: Bijective Transformation and Simplification of YANG to UML*. In: Proceedings of the 10th International Conference on Network and Service Management, CNSM 2014, 17-21 November 2014, Rio de Janeiro, Brazil. pp. 300-303.

- R. Hofstede, L. Hendriks, A. Sperotto, A. Pras. *SSH Compromise Detection using NetFlow/IPFIX*. In: ACM Computer Communication Review, 44 (5). pp. 20-26.
- R. Sadre, A. Sperotto, N. Brownlee, R. Hofstede. *Special Issue of the International Journal on Network Management (IJNM) on "Flow-based Approaches in Network Management: Recent Advances and Future Trends" – Editorial*. In: International Journal on Network Management (IJNM), 2014, pp. 308-309.
- R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, A. Pras. *Flow Monitoring Explained: From Packet Capture to Data Analysis with NetFlow and IPFIX*. In: IEEE Communications Surveys & Tutorials, 16 (4). pp. 2037-2064.
- M. Golling, R. Hofstede, R. Koch. *Towards Multi-layered Intrusion Detection in High-Speed Backbone Networks*. In: Proceedings of the NATO CCD COE 6th International Conference on Cyber Conflict, CyCon 2014, 3-6 June 2014, Tallinn, Estonia. pp. 191-206.
- L. Hendriks, R. Hofstede, A. Sperotto, A. Pras. *SSHCure: SSH Intrusion Detection using NetFlow and IPFIX*. In: TERENA Networking Conference 2014, 19-22 May 2014, Dublin, Ireland.
- R. Hofstede, V. Bartoš, A. Sperotto, A. Pras. *Towards Real-Time Intrusion Detection for NetFlow and IPFIX*. In: Proceedings of the 9th International Conference on Network and Service Management, CNSM 2013, 15-17 October 2013, Zürich, Switserland. pp. 227-234.
- I. Drago, R. de Oliveira Schmidt, R. Hofstede, A. Sperotto, M. Karimzadeh Motallebi Azar, B.R.H.M. Haverkort, A. Pras. *Networking for the Cloud: Challenges and Trends*. In: PIK – Praxis der Informationsverarbeitung und Kommunikation, 36 (4). pp. 207-214.
- P. Celeda, P. Velan, M. Rabek, R. Hofstede, A. Pras. *Large-Scale Geolocation for NetFlow*. In: Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management, IM 2013, 27-31 May 2013, Ghent, Belgium. pp. 1015-1020.
- R. Hofstede, I. Drago, A. Sperotto, R.Sadre, A. Pras. *Measurement Artifacts in NetFlow Data*. In: Proceedings of the 14th International Conference on Passive and Active Measurement, PAM 2013, 18-19 March 2013, Hong Kong, China. pp. 1-10. Lecture Notes in Computer Science 7799 – **Best Paper Award**
- L. Hellemons, L. Hendriks, R. Hofstede, A. Sperotto, R. Sadre, A. Pras. *SSHCure: A Flow-based SSH Intrusion Detection System*. In: Proceedings of the 6th International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2012, 4-8 June 2012, Luxembourg, Luxembourg. pp. 86-97. Lecture Notes in Computer Science 7279 – **Best Paper Award**
- R. Hofstede, A. Pras. *Real-Time and Resilient Intrusion Detection: A Flow-Based Approach*. In: Proceedings of the 6th International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2012, 4-8 June 2012, Luxembourg, Luxembourg. pp. 109-112. Lecture Notes in Computer Science 7279.
- R. Hofstede, A. Pras. *Real-Time and Resilient Intrusion Detection: A Flow-Based Approach*. In: TERENA Networking Conference 2012, 21-24 May 2012, Reykjavik, Iceland.

- R. Hofstede, I. Drago, A. Sperotto, A. Pras. *Flow Monitoring Experiences at the Ethernet-Layer*. In: Proceedings of the 17th EUNICE Workshop on Energy-Aware Communications, 5-7 September 2011, Dresden, Germany. pp. 129-140. Lecture Notes in Computer Science 6955.
- R. Hofstede, I. Drago, G.C. Moreira Moura, A. Pras. *Carrier Ethernet OAM: An Overview and Comparison to IP OAM*. In: Proceedings of the 5th International Conference on Autonomous Infrastructure, Management and Security, AIMS 2011, 13-17 June 2011, Nancy, France. pp. 112-123. Lecture Notes in Computer Science 6734.
- R. Hofstede, I. Drago, A. Sperotto, A. Pras. *Ethernet Flow Monitoring with IPFIX*. In: TERENA Networking Conference 2011, 16-19 May 2011, Prague, Czech Republic.
- R. Hofstede, A. Sperotto, T. Fioreze, A. Pras. *The Network Data Handling War: MySQL vs. NfDump*. In: 16th EUNICE/IFIP WG 6.6 Workshop on Networked Services and Applications – Engineering, Control and Management, 28-30 June 2010, Trondheim, Norway. pp. 167-176. Lecture Notes in Computer Science 6164.
- A. Pras, A. Sperotto, G.C. Moreira Moura, I. Drago, R.R.R. Barbosa, R. Sadre, R. de Oliveira Schmidt, R. Hofstede. *Attacks by "Anonymous" WikiLeaks Proponents not Anonymous*. In: Technical Report TR-CTIT-10-41, Centre for Telematics and Information Technology University of Twente, Enschede.
- R. Hofstede, T. Fioreze. *SURFmap: A Network Monitoring Tool Based on the Google Maps API*. In: 2009 IFIP/IEEE International Symposium on Integrated Network Management, IM 2009, 1-5 June 2009, Long Island, New York, USA. pp. 676-690.