FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Zeek system extension for unirec output
Student:	Matyáš Lhota
Supervisor:	Ing. Karel Hynek, Ph.D.
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

Study computer network monitoring approaches. Get acquainted with Zeek [1] and NEMEA [2] systems and ipfixprobe [3] flow exporter. Implement Zeek plugins, capable of exporting the same extended flow statistics as ipfixprobe into the NEMEA system using unirec interface. Test the developed plugins for correctness and measure the flow statistics extraction speed.

[1] https://zeek.org
[2] T. Cejka, V. Bartos, M. Svepes, Z. Rosa and H. Kubatova, "NEMEA: A framework for network traffic analysis," 2016 12th International Conference on Network and Service Management (CNSM), Montreal, QC, Canada, 2016, pp. 195-201, doi: 10.1109/CNSM.2016.7818417.
[3] https://github.com/CESNET/ipfixprobe/tree/master

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 10 January 2024 in Prague.

Bachelor's thesis

ZEEK SYSTEM EXTENSION FOR UNIREC OUTPUT

Matyáš Lhota

Faculty of Information Technology
Department of Information Security
Supervisor: Ing. Karel Hynek, Ph.D.
May 14, 2024

Citation of this thesis: Lhota Matyáš. *Zeek System Extension for Unirec Output.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

I express my deepest gratitude to my thesis advisor, Ing. Karel Hynek, Ph.D, for his expertise and insightful guidance. I am particularly grateful for his approachable and supportive demeanor, which made our interactions instructive and enjoyable.
This endeavor would not have been possible without Arne Welzel, a core developer of Zeek, who provided ongoing support throughout the plugin's development. I would also like to thank Pavel Šiška, author of the NEMEA framework, whose insights were instrumental in exploring the NEMEA libraries.
Lastly, I must extend my heartfelt thanks to my family for their unwavering support and encouragement throughout my academic journey.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on May 14, 2024

Abstract

This thesis presents the design and implementation of a Zeek extension that enables organizations with existing Zeek implementations to leverage the capabilities of the NEMEA system. Despite Zeek's comprehensive network monitoring capabilities, compared to NEMEA, it does not natively support Python, which can be used for machine learning analysis. To overcome such shortcomings, the project introduces a plugin that facilitates the export of extended flow statistics to NEMEA, similar to the IPFIXprobe flow exporter. The main challenge is integrating the extension with Zeek's core C++ API and NEMEA framework libraries, which is necessary for an effective individual packet analysis and data export in UniRec format. Extensive testing ensures the extension provides accurate data without disrupting Zeek's overall performance. By introducing this extension, Zeek users can now seamlessly leverage all of NEMEA's capabilities and enhance their security posture.

Keywords computer network, network security monitoring, Zeek, NEMEA, IPFIXprobe, plugin development, network flow export, flow-based monitoring, Zeek script

Abstrakt

Tato práce představuje návrh a implementaci rozšíření systému Zeek, které umožňuje organizacím provozující systém Zeek rovněž využívat funkce systému NEMEA. Přestože je Zeek schopen komplexního síťového monitoringu, oproti NEMEA systému například nenabízí podporu jazyka Python, který je velmi populárním pro pokročilou detekci hrozeb za pomoci strojového učení. Tento projekt překonává nedostatky systému Zeek pluginem, který exportuje rozšířené flow statistiky do systému NEMEA, podobně jako IPFIXprobe flow exporter. Největším úskalím je integrace pluginu s C++ API Zeek systému a knihovnami NEMEA frameworku, což je nezbytným krokem pro efektivní analýzu dat a jejich následný export. Důsledné testování zaručuje správnost generovaných dat a neovlivněný výkon celého systému. Uživatelé Zeeku mohou za pomoci tohoto rozšíření jednoduše využívat schopnosti NEMEA systému a zlepšit tak svou bezpečnostní strategii.

Klíčová slova počítačová síť, bezpečnostní monitorování sítě, Zeek, NEMEA, IPFIXprobe, vývoj pluginu, flow export, flow-based monitorování, Zeek skript

Abbrevations

API	Application Programming Interface
APT	Advanced Persistent Threat
BiF	Built-in Function
C2	Command and Control
CESNET	Czech Education and Scientific Network
CSV	Comma-Separated Value
CTU	Czech Technical University
DDoS	Distributed Denial of Service
DPI	Deep Packet Inspection
ICSI	International Computer Science Institute
IDS	Intrusion Detection System
IEs	Information Elements
IETF	Internet Engineering Task Force
IPC	Inter-Process Communication
IPFIX	IP Flow Information Export
IPS	Intrusion Prevention System
JSON	JavaScript Object Notation
LBNL	Lawrence Barkley National Laboratory
MPLS	Multiprotocol Label Switching
NCSA	National Center for Supercomputing Applications
NTA	Network Traffic Analysis
OS	Operating System
RAM	Random Access Memory
RTT	Round-Trip Time
SCTP	Stream Control Transmission Protocol
SIEM	Security Information and Event Management
SLA	Service Level Agreement
TAP	Test Access Point
TRAP	Traffic Analysis Platform
TSV	Tab Separated Values
XML	Extensible Markup Language

Introduction

Network monitoring is a critical process that has become integral to managing IT infrastructure and high-speed networks. The original network monitoring approach was to inspect each packet individually. This became technically infeasible as both network traffic volume and complexity increased. The lack of processing power and digital storage could only be solved with expensive hardware that was not available to everyone, which later led to the adoption of a new approach, flow-based monitoring. This effective and scalable method aggregates packets into flows for analysis, addressing both mentioned limitations, although flow data repositories can still grow to large sizes. A flow is defined 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" [1]. Flow monitoring is now a widespread approach used for security analysis, to comply with data retention laws, and many others [2].

Currently, many solutions exist for comprehensive network monitoring and flow analysis. Two notable independent systems this thesis focuses on are Zeek [3] and NEMEA [4]. Zeek is a widely adopted open-source tool introducing its own Zeek scripting language, an intuitive language focused on network connection and protocol analysis. NEMEA's community and ecosystem, on the other hand, is not too extensive which poses a bigger challenge during setup. However, its Python support naturally offers machine learning integration, which is becoming a necessary part of a well-rounded network traffic analysis approach. To facilitate a seamless connection to the NEMEA system for Zeek users while avoiding the integration of redundant flow exporters, a bridge between the two systems is necessary, which is the aim of this thesis.

In its default configuration, the Zeek system creates multiple connection and protocol-focused logs, none of which contain all flow statistics data necessary for a comprehensive NEMEA communication. Zeek also provides a few output formats, including JSON or TSV. However, NEMEA utilizes UniRec, a minimal binary format for messages, which is not supported. As a modular system, Zeek enables users to extend the core functionality using scripts and plugins, but the problems mentioned above have not yet been addressed. This is why the thesis focuses on the development of a plugin that effectively and seamlessly connects Zeek with NEMEA. The extension will hopefully help organizations using Zeek conveniently leverage NEMEA capabilities and enhance their overall security posture.

The thesis first introduces the evolution and principles of network monitoring to establish a knowledge baseline before exploring Zeek and NEMEA. The two approaches, script and compiled plugin, are discussed in terms of implementation, performance, and convenience. Zeek script provides an intuitive and less technical method to extend the system capabilities but is ineffective in higher throughput environments. The compiled plugin is more suitable for network data processing, but a deeper understanding of Zeek Application Programming Interface (API) and C++ language is necessary. To verify stability and performance, thorough testing is conducted by comparing packet captures and other log outputs and through a network traffic stress test.

Chapter 1
Related Work

1.1 Computer Network Monitoring

Network monitoring is a practice encompassing many activities, many of which are out of the scope of this thesis. In general, it is a practice of continuously collecting information from computer networks with the aim of ensuring it is working properly and securely. The aspects can be simplified into infrastructure health, network performance, and network security. Keeping a network healthy means, for example, keeping all infrastructure up to date — routers, switches, endpoints, and servers. Focusing on optimal performance is crucial to fully utilize the potential of all devices, which, in the end, saves money. The primary focus of this work is the last aspect, network security, which itself is a very broad area. Other than detecting anomalies, Distributed Denial of Service (DDoS) attacks, or data exfiltration using a flow monitoring approach, on which we will focus, keeping a network safe means also detecting malware, intrusion, and many more.

1.1.1 Methods and Techniques

We can classify network monitoring approaches into active and passive [2]. Active means the monitoring device generates traffic in the form of simple pings or more complex queries to either conduct various performance measurements or request data from network infrastructure and endpoints to check their status in terms of proper function and to check for infection. The second approach is more relevant to this work. Passive monitoring is an approach where no additional information enters the networks, and we only listen. It can be referred to as Network Traffic Analysis (NTA) and is often conducted by a more complex and comprehensive network management software conducting real-time analysis, which we will discuss later.

The purpose of NTA is to have an idea of what is happening on our network. This is necessary not only to detect malware but also active devices on the network, vulnerable protocols, and so on. Keeping the collected data is helpful for forensic purposes in case of an incident and sometimes even for complying with various data retention laws.

Two main sources of data are packet data and flows. The inspection of individual packets and their payloads is referred to as Deep Packet Inspection (DPI) and can be collected using, for example, a network switch SPAN port, port mirror, or a network Test Access Point (TAP) [5]. The most significant benefit of DPI is a 100% visibility to the network, however, it can be very resource intensive, especially in high-speed networks where millions of packet can pass through every second. The Flow monitoring approach, on the other hand, usually only collects pieces of every packet, speeding up the collection process and dramatically reducing the volume of data for analysis. The disadvantage of the flow approach is that it lacks the richer detail necessary

to detect certain complex cyber security issues. The collection of flows is commonly facilitated using network infrastructure like routers, which usually integrate flow-collecting mechanisms out of the box.

1.2 Flow-Based Monitoring

During the decade of 1990, the internet and digital communications usage exploded, significantly contributing to the volume of data traveling through networks worldwide. At the same time, technological advancements in network infrastructure, like the transition from dial-up to broadband and later fiber, led to much faster networks, increasing also the speed at which the data needed to be handled. As the network usage grew, so did the variety and frequency of network security threats. Malware distribution, data exfiltration, and DDoS attacks had to be somehow addressed. All of these aspects contributed to the development of flow-based monitoring, which effectively addresses many of these obstacles and shortcomings.

1.2.1 History

Flow-based monitoring was first introduced in 1991 by an Internet Engineering Task Force (IETF) work group. However, its debut was unsuccessful due to privacy concerns and philosophies at the time. The incentive was then resuscitated after 1995, going through a notable development, but due to a lack of vendor interest, the research was again canceled in 2000.

In the meantime, Cisco has been developing a NetFlow technology, initially focused on switching, that is pretty similar to flow monitoring. It implemented a simple unidirectional flow cache, where the flow was defined by an ingress interface, source and destination IP address and port number, IP protocol number, and IP Type of Service. The forwarding decision of each frame was only conducted for the first packet of a flow. All the following packets were automatically switched in the same data plane (also referred to as the “forwarding plane”).

Multiple versions of Cisco NetFlow have been developed through the years, although some of the initial versions were only internal. The two most notable versions are v5 [6] and v9 [7], each introducing significant features and improvements. Version v5 is the most widely supported and considered a standard format for many network monitoring tools. It is fixed in structure, which means it lacks the flexibility to support, for example, IPv6 protocol and cannot be customized. The next significant version, v9, introduced a more flexible and extensible framework to support more flow record formats, including IPv6, Multiprotocol Label Switching (MPLS), and multicast.

Although there is a version referred to as NetFlow v10, it was developed by IETF in an effort to standardize a flow export protocol. From 2004 to 2008, the IP Flow Information Export (IPFIX) IETF working group extended the NetFlow v9 by introducing support for variable-length fields, new Information Elements (IEs) to capture more detailed data about flows, and the option for more flexible and secure transport protocols, including Stream Control Transmission Protocol (SCTP), in addition to UDP and TCP supported by NetFlow v9. In 2014, IETF standardized this NetFlow extension in RFC 7011 as the IPFIX Internet Standard[1]. Since then, IPFIX has become widely supported by routers, switches, and other devices.

1.2.2 Flow Definition and Architecture

According to RFC 7011 [1], flows 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.” These common properties differ between protocols and versions; however, they mostly include packet header fields such as source and destination IPs and port numbers, packet contents, and other metadata.

1.2.3 Architecture

According to Flow Monitoring Explained paper [2], the flow monitoring process architecture is mainly conceptual, comprising 4 phases, which do not have to be completely distinct in reality. The first phase is "packet observation", which describes how packets are read from various media. The second phase called "flow metering and export", focuses on aggregating packets into flows, flow record creation, and their subsequent export. These two phases usually occur on a single device, referred to as a "flow exporter". The next phase is the "data collection", which is concerned with the pre-processing of collected flow records and their optimal storage. The last conceptual phase is "data analysis", where data is parsed to generate simple statistics and alerts and analyzed to detect threats and anomalies. Besides, the data can also be used for network performance measurements. Although each phase can be conducted by a separate machine or tool, many comprehensive solutions like Zeek are capable of all the mentioned steps out of the box, making the whole flow monitoring setup process much more approachable to smaller organizations or teams without the resources for multiple specialized devices.

1.2.3.1 Packet Observation

In the first phase, packets are captured from an observation point, which can be placed not only in a wired or wireless network but also in a virtual one. In the case of a wireless observation, capturing packets on a single point like a WLAN controller is suggested rather than from a wireless device. This approach is easier to implement since decrypting the link layer encryption can be avoided. Most comprehensive software-based network monitoring solutions also support offline input in the form of packet capture (pcap) files. On a typical system, observation is conducted using packet capture libraries capable of packet capture and filtering and offline pcap read and write. Both software and hardware optimizations and acceleration have been developed to speed up this process, mainly through efficient memory management and low-level hardware interaction. After packets are read from the source, they go through multiple pre-processing steps, which are, in part, necessary for further processing.

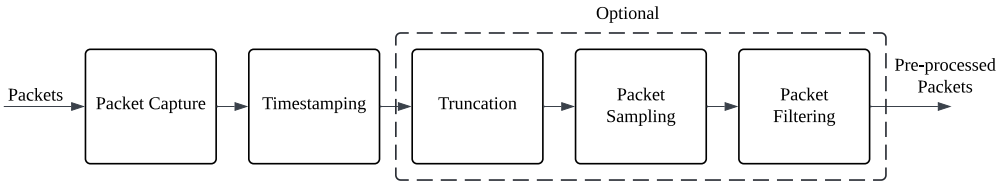

Figure 1.1 Architecture of the packet observation stage [2]

The packet observation phase can be further divided into several steps, as can be seen in Figure 1.1. As mentioned, packets must first be captured or read from a media. The second mandatory step is timestamping, which is, in an ideal case, closely connected to packet capture. Hardware timestamping conducted by specialized network cards is much more accurate than software-based timestamps, which can introduce a slight latency. Timestamping is necessary not only to keep a correct order of events in case of merging data from multiple sources but also for precise analysis and forensic purposes.

The next steps, truncation, packet sampling, and packet filtering are not mandatory, however, in certain situations, they can dramatically save resources and make the whole processing more effective. Truncation of packets is certainly considerable in a flow monitoring environment, as most protocols only work with the packet headers. However, some flow records can be extended with application-level information, in which case the truncation step must be skipped. When the flow monitoring setup cannot process all packets, or in the case of a single packet flow approach, the processed packet number can be reduced by implementing sampling. Sampling

simply discards packets either systematically or randomly, effectively reducing bandwidth and memory consumption. However, it results in information loss, and if not conducted properly, our monitoring results can be biased. When only certain traffic is in our interest, packets can be filtered by selecting only packets having certain properties or by hashing their parts and comparing them with hashes of the desired patterns. This way, uninteresting packets are discarded, and bandwidth and memory consumption are reduced.

1.2.3.2 Flow Metering and Export

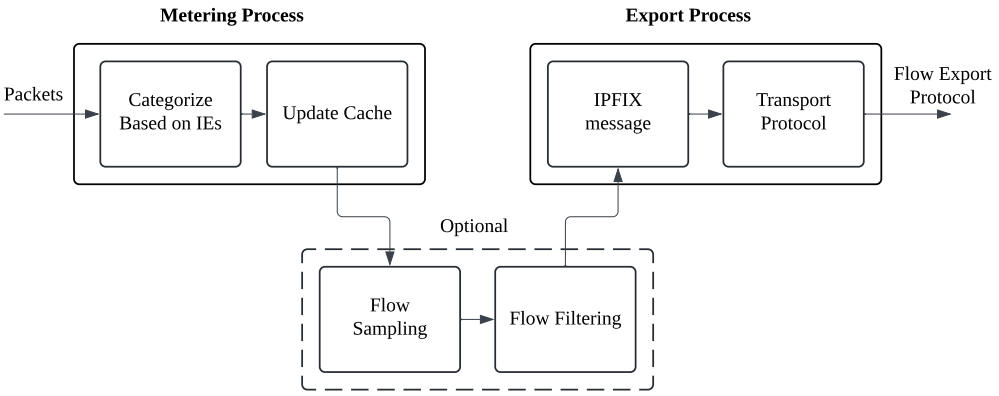

Figure 1.2 Architecture of the Flow Metering & Export stage [2]

After pre-processing, packets enter the metering phase (see Figure 1.2), where they are first aggregated into flows based on IEs defining the flow layout and are later exported in the IPFIX flow records. The IE layout consists of a name like "sourceIPv4Address", ID, a description like "IPv4 destination address in the packet header", and other fields including length. Additionally, to IP addresses, other IEs supported by most exporters can include flow start and end times, protocol identifiers, number of bytes or packets, and information from link and application layers, although commonly focused on network and transport layers. The format of IEs is specified using IE templates, which are propagated from flow exporters to collectors.

Each time a new flow is detected, an entry consisting of IEs is created in flow cache tables to keep track of active flows. These entries are identified by a "flow key", which is defined by a subset of the record IEs, such as IP addresses and port numbers. Flows can be collected either unidirectionally or bidirectionally. In the case of bidirectional metering, there is a distinction between the originator and responder, and the host IEs have to be reversed to match the corresponding flow cache entry correctly.

There can be multiple reasons for a flow to leave the cache table, such as timeout based on total duration or inactivity. The specific timeout values should be correctly configured to achieve the desired length and number of flows. Another reason for a flow entry expiration is a natural connection expiration, such as a TCP FIN message. This is often the desired outcome since the entry includes complete communication between the endpoints. A less desired reason is a cache table becoming full. Since there are many approaches to flow collection, such as a low packet sampling rate, it is necessary to correctly dimension the cache tables to not lose valuable data due to insufficient space.

Before the records are exported, another sampling and filtering can be conducted on the whole flow records instead of individual packets. This is done to reduce the processing requirements of the subsequent phases. The sampling can again be done systematically or randomly while filtering can target specific hosts, subnets, or port numbers.

When the records leave the flow cache, they can be encapsulated in a message and exported

to one or more flow collectors using a flow export protocol. The standard export protocol, IPFIX, uses message formats similar to the one depicted in Table 1.1. Some of the fields are fixed length (numbers in parentheses indicate byte length), while the rest can contain variable length data. The variable part comprises sets of one or more records, either describing an IE template or carrying flow records or metadata for flow collectors. A flow record is then defined in RFC 7011 as "information about a specific flow that was observed at an observation point"[1].

Table 1.1 IPFIX message format [2]

Version Number (2)	Length (2)
Export time (4)	
Sequence number (4)	
Observation domain ID (4)	
Set ID (2)	Length (2)
Record 1	
Record 2	
Record n	

When it comes to the message-carrying protocol itself, IPFIX is protocol-agnostic, typically using SCTP, TCP, or UDP for transport. The suggested option is SCTP, thanks to its flexibility and reliability. However, since there is usually a lack of support from both Operating Systems (OS) and network devices, it can be difficult to implement. [2].

Because of the large processing requirements of the flow exporters, software-based solutions are often not sufficient to handle high-speed network traffic. To overcome this shortcoming, hardware-based or hardware-accelerated solutions are often implemented. Each approach presents a trade-off: the decision lies between speed at a high cost and flexibility with more affordability. There is a variety of comprehensive flow export solutions to choose from, both commercial and open-source such as IPFIXprobe (see Section 1.5).

1.2.3.3 Data Collection

Multiple flow collectors can collect messages exported from flow exporters. These devices or processes are concerned with first receiving data and then storing and pre-processing it before analysis. The main decision regarding storage is between volatile and persistent options. Working with data only in memory comes with a significant speed advantage, but the capacity might be insufficient for processing large volumes of data. While solutions such as NEMEA (which is described in Section 1.4) rely fully on this approach, a combination of the two is often inevitable. Persistent disk storage provides much bigger capacity, but frequent writes can pose a performance challenge. Pre-processing data by compression and correct storage format can make this process less problematic. For example, binary storage format (flat files) introduces much less overhead than databases, but it can be much more complicated to work with regarding filtering and queries. Other than compression, pre-processing data can also include tasks like anonymization, although it is less necessary than in methods like DPI. Similarly to flow export, there is an abundance of both commercial and open-source tools for data collection [2].

1.2.3.4 Data Analysis

The last step in the flow monitoring process is data analysis. While manual analysis is certainly possible and even necessary in some cases, some automation is inevitable to react to incidents promptly in real time. The most basic analysis can include browsing and filtering flow data, generating statistical overviews, and simple reporting and alerting of events such as suspicious numbers of connections or traffic spikes.

A more complicated analysis has to be conducted to identify threats in the network. A forensic approach — focusing on who talked to who, for how long, how much. . . — can help uncover attacks such as DDoS, worms, botnet communication, SPAM, network scans, and even Advanced Persistent Threats (APTs) [2]. Command and control (C2) and other malicious addresses are identified with the help of address reputation lists or blacklists, which can be managed locally or by a third party, such as threat intelligence organizations. Another threat detection approach is called pattern analysis, which is concerned with patterns such as rapidly repeated unsuccessful logins, indicating a brute-force attack.

In addition to threat detection and data filtering, data can be analyzed to monitor the performance and status of active services on the network. This can include metrics like Round-Trip Time (RTT), jitter, delay, packet loss, and bandwidth use. A very practical use of performance monitoring is, for example, service provider Service Level Agreement (SLA) verification, which is most commonly concerned with availability. Performance can be monitored by post-processing IEs exported by flow exporters, which is an easy-to-deploy but not comprehensive solution. A more sophisticated method is to extend the flow exporter data by performance metrics, which provides more information but requires some configuration and relies on higher protocols. Again, one can choose from multiple open-source and commercial tool options for data analysis, such as NEMEA [2].

1.3 Zeek

Zeek, which was originally called Bro, is a widely adopted, passive network traffic analyzer[3]. It is an open-source software with a big community behind it. Its primary use is identifying vulnerable SW and malware in computer networks, brute forcing, web applications, and generally suspicious or malicious traffic. Zeek is so widely used because it is able to run on commodity HW, which presents a low-cost solution for users of all types.

1.3.1 History

Zeek was initially created by Vern Paxson in an academic research at Lawrence Barkley National Laboratory (LBNL). Although the project received much support from the International Computer Science Institute (ICSI) in 2003, it wasn't suitable for a wider audience because of its complexity and generally bad end-user experience. In 2012, the National Center for Supercomputing Applications (NCSA) stepped in and helped develop the Zeek 2.0 version. This version introduced a much better user interface which helped Zeek spread in the networking community. An online educational platform was also introduced for developers to get hands-on experience with Zeek's custom scripting language. Around 2019, a new version, 3.0, was introduced, finally under the name Zeek.

1.3.2 Usage and Capabilities

As mentioned, Zeek is mainly used as a security monitor to help with network incident investigations; however, it can also be leveraged for troubleshooting and performance measurements. Zeek reports in the form of various logs, each focused on a specific area of NTA. These can include protocol or flow-focused logs, logs of files extracted from network communications, and so on. The information Zeek reports can then be formatted and further analyzed by Security Information and Event Management (SIEM) solutions.

The tool is fully customizable, meaning one does not have to rely on Zeek's default functionality and can introduce custom scripts or plugins. Zeek developed its own Turing-complete, domain-specific scripting language, which was developed with network analysis in mind. No

functionality is hard-coded into the Zeek core, which means users have full control over Zeek's functionality.

Despite Zeek being a versatile tool, it is not optimized for all the aspects of a comprehensive NTA. Although it can be used as a signature-based IDS, or as a protocol analyzer, it is not optimized for byte matching and packet-level inspection. For these purposes, users should turn to other specialized tools such as Suricata, which is more suitable for DPI thanks to its multi-threaded architecture [8]. Zeek mainly specializes in semantic misuse, anomaly detection, and behavioral analysis.

Data sources in securing systems can be generally classified into 4 distinct categories. [3] The first data source that can be leveraged is third-party data from either law enforcement or non-government threat intelligence organizations. The second source is infrastructure and app data, which can be collected, for example, from cloud logs. The next sort of data is endpoint data collected from devices (endpoints) connected to a network, which includes data like system logs and system configuration. The last category is data collected directly from the network, which can be further categorized into four types: full content, extracted content, alert data, and transaction data — the latter being Zeek's main specialization. Although it is capable of extracted data analysis and alert handling, there are other tools analysts should rather refer to. When it comes to full content analysis, Zeek has not been optimized for writing full traffic to disk, which means using it for this purpose can be inefficient, resulting in data loss.

Thanks to its versatility and low-cost implementation, Zeek is now used in various environments, including universities, research labs, supercomputing centers, corporations, and government agencies. The tool can function in high-speed, high-volume networks without any difficulties if no resource-intensive functionality is added to the system. When a cluster setup is correctly implemented, it can be used on links going up to 100Gbps.

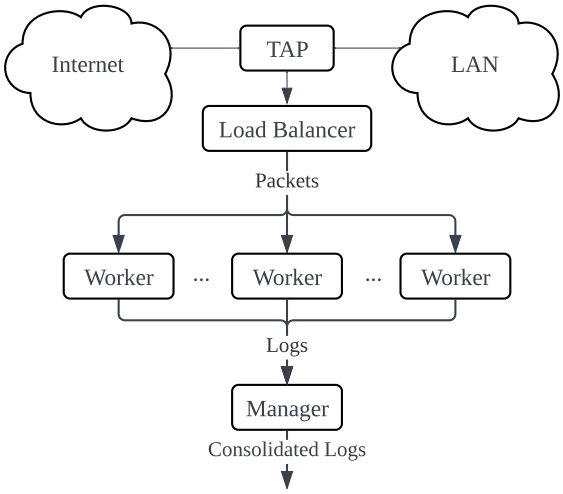

Figure 1.3 Zeek cluster setup architecture [3]

Since Zeek is not multi-threaded, the limitations can be overcome by implementing a cluster setup, which is supported by default. The cluster setup architecture can be seen in Figure 1.3. The traffic is simply split by a load balancer, which distributes the work between multiple workers or threads (Zeek instances). The whole process is overseen by a central manager system, which consolidates the individual logs into a single log. Both single and multi-system setups are supported out of the box.

1.3.3 Architecture

On the highest level, Zeek consists of two main components: the event engine, which reduces the packet stream to higher-level events, and the policy script interpreter, which executes event handlers defined in Zeek scripts. The simplified architecture can be seen in Figure 1.4.

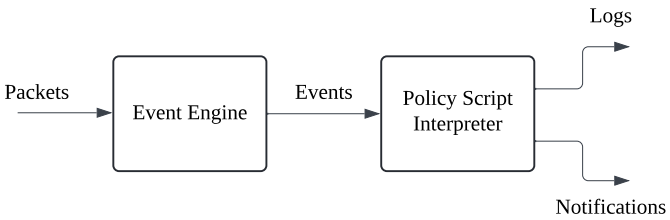

Figure 1.4 Zeek high level architecture [3]

The event engine can generate events in all phases of the packet processing pipeline. The first phase is the input source, where events such as `raw_packet` are generated. Going up through the network layers, packets are first analyzed individually, providing events from the data link layer. Further down the processing pipeline, Zeek generates session-related events and extracted files. Thanks to Zeek's plugin architecture, the core functionalities can be extended in any of these stages.

When generated, an event only provides information that something happened, not why it happened. The analysis of "why" is conducted using Zeek scripts, which react to the events generated by the event engine. All of the analysis and output functionality Zeek offers, both user-defined and default, is, in fact, implemented in the form of Zeek scripts, which are interpreted by the policy script interpreter. The scripts can maintain state over time, which means analysis of observations across connections and hosts is possible, such as a series of failed login attempts from the same IP address within a short period indicating a brute-force attack. These scripts, however, can be further enhanced by compiled components for more complex and effective functionality.

1.4 NEMEA

While not so widely adopted, NEMEA is another invaluable open-source network traffic analysis system. It was developed by the Czech Education and Scientific NETwork (CESNET) research team with support of Czech Technical University (CTU) and introduced at a conference in Montreal in 2016. It is a stream-wise, flow-based system consisting of many independent modules, each with its own task, making the system very flexible and scalable. The modules are interconnected by communication interfaces, between which messages are sent in NEMEA's custom UniRec binary data format. The individual messages can contain either flow records, alerts, statistical data, or pre-processed data [9].

1.4.1 Capabilities

Because simple flow analysis is insufficient for successfully identifying modern-day attacks, the NEMEA system also integrates application layer (L7) [10] data analysis. Its stream-wise architecture means it analyzes data almost completely in memory, effectively avoiding slow persistent storage disk read and write operations. The system is capable of online and offline data analysis, which can be very convenient for the new traffic analysis module development and testing. Thanks to NEMEA's Python support, the module development is accessible compared to systems relying only on low-level languages like C. Python also introduces a vast array of libraries

for data analysis and machine learning, which can notably simplify the development of advanced threat detection mechanisms.

From the start of development, NEMEA was intended for use with high-speed networks, easily capable of handling 100GE links. To imagine its performance in a real example, CESNET shared information about a NEMEA setup they have deployed and its capabilities. The instance was deployed on a single server with six CPU cores and 12GB of Random Access Memory (RAM). On average, while utilizing about 40% of the available resources, the system has detected approximately 110 000 horizontal portscans, 12 000 SSH brute force attacks, and 2400 DDoS attacks every day [9].

1.4.2 Usage

Security analysis is often done using an Intrusion Detection System (IDS) or Intrusion Prevention System (IPS), which have the capability of both individual packet and flow processing like Zeek (described in Section 1.3). NEMEA, on the other hand, is strictly flow-based, which not only improves the performance by reducing the processed data but can also facilitate the detection of different kinds of attacks.

Every NEMEA instance can be unique since each user can build different data paths using NEMEA modules — each performing different tasks, underlying its flexibility. Modules are connected in the form of trees — directed acyclic graphs — where one module usually serves as a main ingress point. This module gathers or creates flows and sends them to the NEMEA system. On the other side of the system lie the end modules, which are commonly concerned with either logging or alert management.

Some of the module capabilities used in a typical system can include detection up to transport layer [10], which is capable of identifying port scanning and attacks like DDoS, while application layer-focused modules can specialize in analysis of protocols like DNS, SSL, or SIP. Other modules can focus on infected device detection by extending flow records with fields such as URLs, which can be used for suspicious server or device identification. The end modules handling alerts are called “reporter” modules and convert alerts from detectors into a unified format to be either logged, stored in a database, sent by email, or sent further to other systems specialized in alert collection. Other commonly used modules are modules for offline testing. These modules can read offline data formats like nfdump format, fastbit DB used by IPFIXcol flow collector, or Comma-Separated Value (CSV) format. In addition to simple offline reading, these modules are capable of “store and replay”, which can be an invaluable functionality for repeated module testing.

The whole system can be easily reconfigured, which is useful not only in production environments but also for developing new traffic analysis methods. A significant advantage is that modules can be added or removed at run-time, enabling safe system modification in critical environments. As NEMEA is developed in C, C++, and Python programming languages, it can be easily deployed on any UNIX-like OS. Additionally, thanks to the modular architecture, it is not limited to single server setups. NEMEA can be distributed on multiple hosts, where modules can communicate over the network.

1.4.3 Architecture

It has been mentioned that NEMEA systems consist of multiple modules, but other resources are also necessary for them to function correctly and effectively. Since all modules typically share some kind of functionality, like communication interfaces, these common features are implemented in a set of libraries referred to as the NEMEA framework. Additionally, a supervisor process is introduced to the system, controlling and monitoring all other modules.

It was mentioned that modules are connected in the form of trees. Modules can be hence described as unidirectional building blocks of the NEMEA system, receiving a stream of data

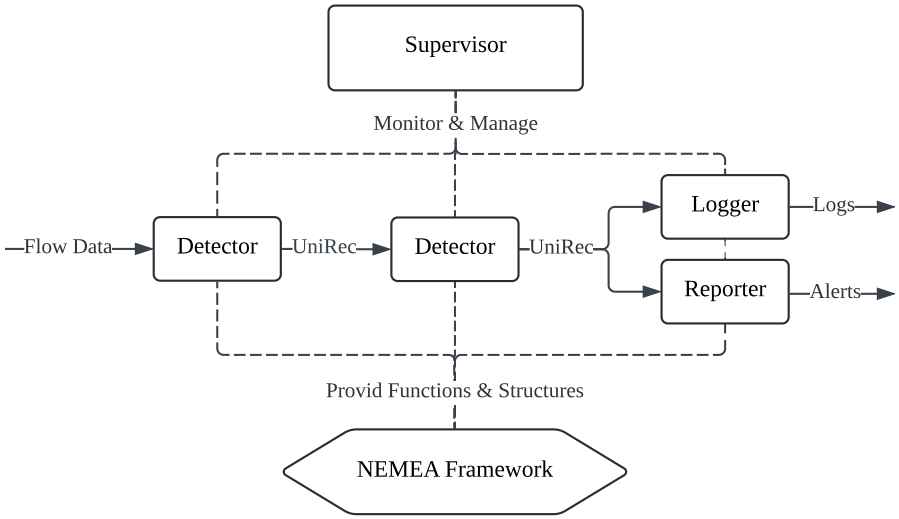

Figure 1.5 NEMEA system architecture

on an input interface, independently processing it, and sending a stream of data to their output interface. A simplified architecture of the NEMEA system can be seen in Figure 1.5, showing an example system of two detector modules and two output modules, all managed by the supervisor, leveraging the NEMEA framework libraries. Detector modules are typically positioned as branches in the tree structure, having a clear task of flow data analysis and identification of malicious traffic such as port scanning. Other modules can have various purposes, such as the leaf modules (like logger and reporter in the diagram) being tasked with logging, storing, and reporting. In contrast, other branch modules can work on data post-processing and pre-processing like filtering, aggregation, or merging.

The NEMEA framework is an essential part of the system, as it implements a Traffic Analysis Platform (TRAP) library, which defines communication interfaces and functions. The interfaces are unidirectional, capable of processing messages of size up to 64kB. The interfaces can facilitate Inter-Process Communication (IPC) on a single host by leveraging UNIX domain sockets, network communication between individual hosts using TCP sockets, writing and reading data streams from files, or conveniently discarding data in what NEMEA refers to as a "black hole."

The framework also defines the key message data format for effective communication between modules, UniRec. This binary format is very similar to C structs in terms of fast access, but it additionally supports variable-length fields, and its template can be defined at run-time. When two interfaces are connected, the sending and receiving formats are checked, and if there is a match, a connection is established. Other than UniRec, NEMEA further supports the JSON message format and unstructured data, although their use is not very common.

Modules like detectors usually overlap to some extent in their use of functions and data structures, so the NEMEA framework includes a common library. This library provides effective structures like trees, Bloom filters, hashtables, and various hash functions.

For the whole system to function properly, a supervisor process manages all active modules. The management is conducted using an Extensible Markup Language (XML) configuration file, which is changeable at run-time. The supervisor first retrieves information from the modules, compares it to the provided configuration, and performs action in case of discrepancies, enforcing the configuration. This control is conducted periodically, which enables the run-time changes.

1.5 IPFIXprobe

One notable specialized flow exporter is IPFIXprobe. It was developed by CESNET — initially as a standalone NEMEA system module called flow meter — and deployed in production in 2021. It is currently capable of handling 100 Gbps high-speed network lines, with a view to supporting up to 400GE cards in the future. Although IPFIXprobe is not widely adopted, it is used in production monitoring not only by the CESNET association but also by many Czech universities.

The architecture of IPFIXprobe is modular and extendable in the form of plugins. Its structure can be divided into four directional parts, defining the processing pipeline: input, processing, storage, and output plugins. Input plugins facilitate packet observation from sources such as pcap files, FPGA cards, or IPFIXprobe default raw socket input. The exporter can be extended by processing plugins that export various new information from application layer protocols, such as DNS, HTTP, SIP, or NTP. The storage part is essentially a flow cache, creating and updating active flow entries. At the end of the pipeline, there are output plugins that provide several options for the export of flow records. Three notable output formats are IPFIX, UniRec as a data source for the NEMEA system, and human-readable text format.

The most important processing plugin is PSTATS, as the Zeek system extension presented in this work mimics its functionality. The PSTATS plugin receives a maximum element count number parameter, which specifies how many packets should be analyzed by the plugin in each flow. It then creates arrays to record each analyzed packet's length, time, direction, and TCP flags in a flow. It also supports including or excluding zero-length packet information from the arrays. Although the fields are defined as UniRec elements, the equivalent fields are exported in formats like IPFIX [11].

Chapter 2
Analysis and Design

This chapter serves as a foundation for the development of the Zeek extension. It first identifies both functional and non-functional requirements of the extension, which will be necessary later in the development phase. The chapter also explores the existing solutions and their limitations. Based on the extension requirements and shortcomings of the existing solutions, multiple design considerations and decisions are discussed in terms of both implementation complexity and performance. Then the chapter focuses on the detailed design of the extension in two variations: Zeek script and compiled plugin.

2.1 Extension Requirements

Before delving into the development, it is necessary to outline the specific requirements the extension needs to address. Essentially, the goal is for Zeek to act as a flow exporter — reading packets from a source, categorizing them into flows, and subsequently exporting them — sending extended flow data to a specified TRAP output interface like a typical non-leaf NEMEA module, in this case specifically as the main ingress point.

Since Zeek is a network monitoring tool, it can read data from all common sources, so the extension does not have to reimplement any packet observation mechanism. Since it is necessary to collect per-packet statistics (PSTATS) during the flow metering phase, accessing and analyzing each incoming packet individually while updating the appropriate flow records is necessary.

To correctly mimic the functionality of IPFIXprobe's PSTATS plugin, the extension has to accept two parameters. The first parameter is the maximum number of packets to be analyzed in each flow, while the second tells whether empty payload packet statistics are to be recorded as well. These parameters can significantly reduce processing requirements and stored data, as some flows can consist of thousands of packets. In the case of focusing on the initiations of network connections in the analysis phase, one can effectively disregard the rest of the conversations by setting the maximum parameter to a lower number. Although empty packets can be disregarded in cases like TCP ACK messages, which do not carry any significant information for network security, including them in an analysis might be highly relevant for detecting DoS attacks or network scan activities. Additionally, to function as a NEMEA output module, an output interface parameter has to be accepted by the extension for the UniRec messages to be exported.

The next task is to intercept Zeek's processing pipeline at the point where flows are expired from the flow cache / terminated. At this stage, the extension will gather all relevant data and create a UniRec record before exporting to ensure compatibility with other NEMEA modules. The process of single-connection data flow — beginning with accepting categorized packets

from Zeek, through basic and extended flow data collection, to UniRec export — is designed in Figure 2.1.

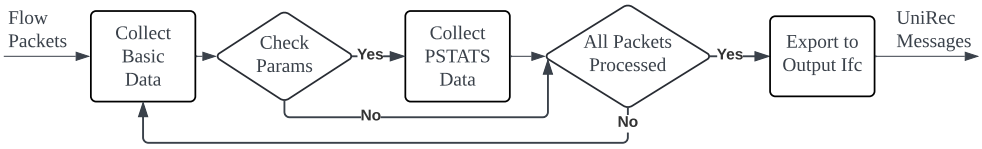

Figure 2.1 High-level extension design

While addressing all the functional requirements, it is also necessary that the extension does not impact Zeek's overall performance, as information loss can have a detrimental effect on the security analysis results. The extension must be reliable, accurate, and secure, which requires proper error detection and correction mechanisms at all development stages. Additionally, to properly function, the extension must be developed with usability in mind, in terms of its installation and integration, and its extensibility by other developers by properly commenting on the code and adhering to best programming practices.

2.2 Existing Solutions

With the aim of avoiding unnecessary reimplementation of existing functionality, let's explore the solutions Zeek might present to cover the functional requirements. To extend its base functionality, Zeek introduced its custom Zeek scripting language. As was already mentioned in Section 1.4, when Zeek processes information collected from the network, it generates system-wide events, which can then be taken care of by Zeek script event handlers. Zeek generates many events in multiple processing phases, which might help with both individual packet and flow processing.

While Zeek script can be very convenient, it is still an interpreted language, which might pose an intolerable overhead in some cases. For this reason, Zeek also provides an option to add any functionality in a compiled form. It provides a C++ API that can be leveraged to register new components, but also to compile individual functions and make them available in the Zeek script environment. In reality, most Zeek plugins are constructed using a combination of the two approaches.

2.2.1 Zeek Script

Zeek's robust scripting language allows users to add custom functionality that responds to network events. When Zeek detects activity on the network — such as a new connection establishment or specific protocol transaction — an event is generated. Zeek scripts can register handlers for these events, which enables them to process data in real time. The language introduces special data types specifically designed for network analysis, including sets, tables, and records that efficiently handle IP addresses, ports, and more.

The simplicity of Zeek's scripting language makes it possible to develop and deploy custom network analysis logic quickly. This is especially important for adapting to new threats or customizing analysis to specific network environments. The possibility of extending Zeek using the scripting language also makes it more accessible for network security professionals without advanced programming skills. Network analysis functions, on the other hand, are typically written in low-level languages to satisfy the environment's huge processing requirements, making them inaccessible to a broader audience. The Zeek community has contributed a large amount of scripts that extend Zeek's core functionality, ranging from detecting sophisticated attacks to integrating with third-party services. An example of an SSH brute-force attack implemented in Zeek script can be seen in Code Listing 2.1.

Code listing 2.1 Example of simplified SSH brute-force attack detection

```
global failed_login_attempts: table[addr] of count = {};

event ssh_auth_failed(c: connection) {
    local src_ip = c$id$orig_h;
    failed_login_attempts[src_ip] += 1;

    if (failed_login_attempts[src_ip] >= 100) {
        print fmt(}Potential SSH brute-force
            attack detected from %s}, src_ip);
        # Take additional actions here, like sending an alert.
    }
}
```

Code listing 2.2 Plugin initiation command

```
$ zeek-aux/plugin-support/init-plugin <plugin-directory-name> \
    <namespace> <plugin-name>
```

Zeek provides a modular architecture, allowing users to share and load scripts dynamically. The scripts can be managed using the `@load` directive, with which other scripts can be loaded. Thanks to this capability, developers do not have to reimplement existing features but can simply import them similarly to other programming languages like C or Python. At the same time, users can dynamically unload modules, which can help keep the system as lightweight as possible.

Although Zeek scripts are an accessible way to extend Zeek's functionality, they must be interpreted at runtime, introducing overhead and performance issues under high network load. This can be especially problematic when scripts perform complicated data processing or when too many scripts are active simultaneously. As the volume of network traffic increases and analysis requirements become more complex, managing a large number of scripts can become challenging. While a script might work well by itself, its introduction to a larger system might cause trouble.

To effectively make use of the Zeek script, it is important to utilize built-in functions and data structures without introducing any unnecessary complexity. While some events are generated only once or twice per connection, other events – like individual packet events – can be very expensive in terms of resources and should not be used in production. Despite being ideal for a wide range of applications, Zeek script has its performance limits. That is why it is often complemented by compiled components.

2.2.2 Compiled Zeek Extensions

When Zeek script solutions come up short in terms of performance, new functionality can be introduced to the system in a compiled form. Zeek is built in C and C++ programming languages, which is a common option for network monitoring as it is very low-level and can process network traffic without a significant overhead. To enable Zeek's extension using a compiled plugin, Zeek introduces a C++ API that developers can leverage to register both whole new system components — protocol analyzers, data loggers... — and individual Built-in Function (BiF) elements for use in Zeek script like functions or events.

A key collection of tools for Zeek compiled plugin development is zeek-aux. In addition to development and log manipulation tools, it contains a script called init-plugin, which generates a template for plugin development using the syntax in Code Listing 2.2

This automation significantly simplifies the creation of new plugins by providing a good

Code listing 2.3 Plugin template generated by init-plugin

```
#include "Plugin.h"

namespace plugin { namespace namespace_plugin-name { Plugin plugin; } }

using namespace plugin::namespace_plugin-name;

zeek::plugin::Configuration Plugin::Configure()
        {
        zeek::plugin::Configuration config;
        config.name = "namespace::plugin-name";
        config.description = "<Insert␣description>";
        config.version.major = 0;
        config.version.minor = 1;
        config.version.patch = 0;
        return config;
        }
```

starting point for development. The plugin skeleton contains a plugin name file marking the directory as containing a Zeek plugin, a directory structure for plugin scripts, and a source directory for BiF and C files already including a minimal plugin structure seen in Code Listing 2.3. To automate and simplify the compilation of the plugin, the skeleton includes a configuration script that prepares the compilation environment. After the configuration is done, developers can simply run `make` to build the plugin, or `make install` to install it.

Zeek's plugin class is a base class for all plugins. It encapsulates all functionality that extends the Zeek subsystems, such as protocol analysis or special format logging. Using the `Plugin` class methods, developers can register new components, perform actions before or after the plugin is loaded, and hook into certain operations and events, such as constructing analyzer trees, incoming unprocessed packets or log writes.

Compiled plugins offer a much better performance, which is critical in high-traffic networks. They operate within Zeek's runtime, executing more efficiently than interpreted script-based solutions. Their deeper integration with Zeek's core through the API allows them to access data with minimal overhead, capturing and analyzing network traffic with minimal delay. In addition to new components, developers can compile individual functions that can be introduced to the Zeek script environment. Their purpose is not limited to network analysis or logging. Such functions can take care of resource-intensive tasks like large array operations or frequent function calls, which would be much more inefficient when implemented in an interpreted script.

While powerful, compiled plugins introduce a much higher level of complexity compared to script-based extensions. To correctly implement a C++ plugin, developers must be proficient in C++ programming and have a good knowledge of Zeek's core environment and its underlying structures. This can definitely steepen the learning curve and complicate the development process.

Since C++ is a very powerful language, it is important for developers to be careful and follow the best practices of software development so as not to disrupt the whole Zeek system. This includes thorough error handling and testing of the plugin — improper memory access can result in program crashes — and properly documenting all code. Adhering to Zeek's coding standards and maintaining comprehensive documentation can ensure the plugins' maintainability and readability for auditing by other developers and users.

In summary, relying solely on a script-based solution is not common when implementing new comprehensive plugins that perform multiple tasks, such as analysis and logging. Although Zeek script is versatile, implementing certain functionality can introduce large overhead, which

Code listing 2.4 Zeek local site policy file excerpt

```
@load misc/loaded-scripts
@load tuning/defaults
@load misc/capture-loss
@load misc/stats
@load frameworks/software/vulnerable
@load frameworks/software/version-changes
@load-sigs frameworks/signatures/detect-windows-shells
@load protocols/ftp/software
@load protocols/smtp/software
@load frameworks/files/hash-all-files
@load frameworks/files/detect-MHR
@load policy/frameworks/notice/extend-email/hostnames
@load frameworks/telemetry/log
...
```

could lead to data loss. For this reason, resource-intensive tasks are generally taken care of by compiled parts of code, significantly enhancing overall performance. Thanks to the zeek-aux toolset, developers can focus on creating security features, knowing that the core intricacies of integration and compilation are well-managed.

2.2.3 Available Plugins and Extensions

Even when run without any extensions enabled, Zeek's base functionality is extensive and capable of many network monitoring tasks. Zeek's core system handles the initial packet capture and basic analysis, including the breakup of traffic into flows, identification of service protocols, and basic connection logging. In its default configuration, Zeek additionally loads the local site policy scripts, adding a more in-depth analysis of specific protocols, file hash comparison, basic signature detection, and statistics and tuning scripts for performance optimization, all of which can be seen in Code Listing 2.4. Since the base analyzers extract a lot of information, the optimal scenario would be to simply collect the desired extended flow information from one or more sources already existing in the system.

2.2.3.1 Connection Object

The most fundamental construct in Zeek is a `connection` object, which encapsulates the state of network connections in a nested record structure [3]. Zeek's core passes it to all events related to a flow, such as `new_connection`, `new_packet`, `file_over_new_connection`, and many more. It is a built-in record type that holds information such as connection ID — a connection identifying 4-tuple of the originator and responder IP addresses and port numbers — based on which packets are categorized into flows, start time and duration, and other connection and host-specific information. When a new module collects additional information related to a connection — especially when the information is being logged — the best practice is to extend the built-in connection record to additionally hold the new information. Figure Code Listing 2.5 shows an example of how Modbus extension defines its own data structure and redefines the `connection` record to include it.

Although there is an abundance of such connection extensions, their focus is mostly on specific protocols like DHCP, DNS, FTP, SSL, or HTTP. Since the extension this thesis develops is concerned with protocol-agnostic flow data, none of these redefinitions are of interest. The `connection` record itself, however, holds some basic flow information that could be extracted with the extended PSTATS data and will be leveraged in the extension. The fact that the record

Code listing 2.5 Connection object redefinition by Modbus extension

```
type Info: record {
    ts:          time         &log;
    uid:         string       &log;
    id:          conn_id      &log;
    tid:         count        &log &optional;
    unit:        count        &log &optional;
    func:        string       &log &optional;
    pdu_type:    string       &log &optional;
    exception:   string       &log &optional;
};

redef record connection += {
    modbus: Info &optional;
};
```

lacks some of the required information means that a new packet-level analysis will have to be conducted to access individual packets' timestamps and payload sizes, determine their direction, and extract TCP flags if relevant. Additionally, the connection record must be redefined to store the extra collected information appropriately.

2.2.3.2 Output

When all necessary information is collected, it must be formatted into UniRec format and exported to NEMEA. By default, Zeek supports two output options: recommended Tab Separated Values (TSV) format and JavaScript Object Notation (JSON) format. Although these formats can be useful for efficient data access or further processing by other tools, their conversion to UniRec format is not a desired solution.

Since other community extensions, excluding NEMEA, focus mostly on third-party tool integration, the extension will need to implement a custom logging mechanism, either in the form of a new logger system component or a dedicated compiled BiF function.

2.3 Detailed Design and Considerations

The development of the Zeek extension to export enhanced flow statistics requires a design approach that addresses the extraction of new data, their efficient handling within Zeek's existing ecosystem, and their extraction in a custom format. The design process begins with a solely script-based solution, leveraging Zeek script's network-focused capabilities. However, as some events are not suggested for use in production and some functionality is more conveniently implemented in C++, certain phases consider a compiled solution.

2.3.1 Defining New Record

To store required data and facilitate logging, the first step is to define a new record type within a new module, `Pstats`. Adhering to Zeek's best practices, an `Info` record will be created to encapsulate all the new data fields necessary to store extended flow data — a simplified design based on modbus extension (Code Listing 2.5) can be observed in Code Listing 2.6.

To define individual fields, Zeek's scripting language provides multiple data types for the storage of network data. In addition to custom types, the script provides several attributes that can be assigned to variables and record fields such as `&optional`, `&default`, or `&log`. The

Code listing 2.6 Initial declaration of the Pstats::Info record type

```
module Pstats;

type Info: record {
    ...
};
```

record has been designed to store data in an appropriate format while addressing all record fields extracted by IPFIXprobe with PSTATS extension:

- **IP Addresses (dst_ip, src_ip)** - These fields will record the source and destination IP addresses to identify the flow endpoints. The `addr` data type represents an IP address and can be used for tasks like effective subnet matching.
- **Port Numbers (dst_port, src_port)** - Another basic flow information is transport-level port numbers. The special `port` data type holds an unsigned integer, which can be further followed by protocol specifications like `/tcp` or `/udp`.
- **Protocol (protocol)** - This field indicates the protocol number, utilizing the `count` data type. Although protocols could be represented with a smaller 8-bit data type, `count` is the only basic unsigned integer type Zeek's script offers.
- **Endpoint bytes (bytes, bytes_rev)** - Two counters aggregating the bytes sent from connection endpoints, the former from originator to responder and the latter in reversed direction. These fields can only be of positive value, hence `count` type has been selected.
- **Times (time_first, time_last)** - The flow initiation and termination times will be stored using the `time` data type. It is a temporal type representing an absolute time. It is essentially a double representing seconds with microsecond precision, which can additionally be used to produce interval data types using simple subtraction.
- **MAC Addresses (dst_mac, src_mac)** - MAC addresses will be stored as simple strings. The `string` data type can easily accommodate the hexadecimal notation of MAC addresses without the need for a more complex structure.
- **Packet Counts (packets, packets_rev)** - These fields will count the packets from the communication originator and responder — same notation as bytes. The choice of the `count` data type here is straightforward, as it is designated to hold non-negative integers. This effectively doubles the maximum possible value the field can hold compared to `int` type.
- **PSTATS Data** - Individual packet data fields will be represented using a `vector` data type capable of dynamically holding multiple values. Timestamps are intuitively kept in a vector of `time`, lengths and flags in a `count` vector, and directions in a vector of `int`, as its possible values are either 1 or -1, representing packet directions from originator and responder respectively.

When a record is defined and meant to be attached to a connection record globally, it is first necessary to encapsulate it in an export block. This enables the record to be visible in other modules using the namespace operator, in this case, specifically `Pstats::`. Once the new record type is available globally, the `connection` can be redefined to include the `pstats` record optionally. It is declared as optional, meaning the `pstats` field can be missing during `connection` instantiation, a necessary step for Zeek to function properly — see Code Listing 2.7 again based on modbus example in Code Listing 2.5.

Code listing 2.7 Export and assign the Pstats::Info record to connection

```
export {
    type Info: record { ... }
};

redef record connection += {
    pstats:    Info   &optional;
};
```

2.3.2 Record Instantiation

For the `Pstats::Info` record to be populated with extracted information, it first has to be instantiated. An appropriate time for its instantiation is at the very beginning of each connection, before its population, during individual packet inspection. Although Zeek offers a few events related to connection initiation, not all are suitable for this purpose. The event `new_connection` was chosen over the `connection_established` event. Despite being raised at the beginning of each connection, the `connection_established` is generated when a SYN-ACK packet is received from the responder, meaning in case a whole handshake is recorded, a SYN packet from the originator precedes the event. On the other hand, `new_connection` is "raised with the first packet of a previously unknown connection"[3]. The extension ensures that the `pstats` record is prepared and attached to every connection from the start, even before the first packet is processed by opting for this event. In any other case, some information might be missed, invalid reads and writes might occur, or a redundant check for `pstats` presence at the `connection` record would have to be conducted with each incoming packet.

When the `new_connection` event is raised, the `connection` record is already instantiated, meaning Zeek has conducted the initial analysis to categorize the first packet into a connection. As Zeek categorizes flows based on the IP address and port number 4-tuple, it is certain that both originator and responder addresses and ports are available and can be used for custom record instantiation. However, other details like the responder MAC address or the number of packets are not yet known, and their extraction has to be handled later in the communication. It is also important to mention that port numbers in ICMP connections are not actually port numbers but rather store the ICMP message type and code in source and destination ports, respectively.

2.3.3 Populating the Record

The initial approach to populating the `pstats` record with packet and flow statistics utilizes solely Zeek's scripting language. Since some basic flow data is readily available at the `connection` record, the main challenge is to access each packet in a flow individually and update appropriate record fields. Although this approach can achieve the desired functionality, packet-level event handlers are strongly discouraged because, in a busy network, the number of packets to be processed can easily become impossible to handle without loss. Zeek's documentation states that packet events are "usually infeasible to handle when processing even medium volumes of traffic in real-time" [3], which means a compiled solution has to be employed to achieve packet data collection without complications in any given environment.

2.3.3.1 Script Design

To leverage Zeek's event-driven architecture, a few events can be considered for the purpose of individual packet access:

- **raw_packet** - The first raw packet event is "generated for every packet Zeek sees that has a valid link-layer header" [3]. As a parameter, this event provides a raw packet header record that holds a layer 2 header [10], and optionally higher level protocol headers like IP header, TCP header, or UDP header if present. Although this event provides a comprehensive insight into all relevant protocol headers, the main issue is the absence of flow categorization. Since this event is very low-level, the categorization has not yet been conducted and would have to be reimplemented. As IP addresses and port numbers are used for categorization, first it would be necessary to determine whether the IP protocol is present, then eventually the information would be extracted. Zeek records flow data in both directions (biflows), meaning the extracted address-port pairs would have to be checked in both orders for flow matching. Furthermore, since `connection` record is typically accessed through event handlers receiving it as a parameter, there is no direct way to look up an active `connection` based on an ID and a more complex structure like a Zeek table (map) would have to be implemented to keep track of active connections. This solution, although functional, introduces an unnecessary redundancy in both flow categorization and memory allocation.

- **tcp_packet** - The next considered `tcp_packet` event is an event unsurprisingly generated for all TCP packets. The decision not to utilize this event is straightforward, as TCP is only one transport protocol out of three protocols tracked by Zeek: TCP, UDP, and ICMP. Using this event, many packets would not be appropriately analyzed, which would result in significant information loss.

- **new_packet** - The most fitting is the `new_packet` event. Most importantly, this event is low-level enough that it is generated for all processed packets — except raw packets that do not pass certain checks — but at the same time high-level enough that the packet has already gone through the flow categorization process, so the event handler provides both a `connection` parameter for flow information and `pkt\_hdr` for packet-specific information. By choosing this event over `raw_packet` and `tcp_packet`, it is ensured that no data is lost in processing and both storage and categorization redundancy are successfully avoided.

As all data in the packet headers is static throughout the analysis compared to time, let's start by addressing timestamping first. The actual order in which PSTATS information is extracted in the `new_packet` event handler is discussed in subsequent design and implementation chapters since it relies, for example, on the extension parameters that are yet to be defined. There are two Zeek built-in functions that can be potentially used for accessing time:

- **current_time** - This function is self-explanatory, as it returns the current time according to the operating system. Use of this function is not recommended for networking uses as it introduces a slight discrepancy between the actual time a packet was observed and the moment the operating system handles the request for the current time. From the moment a packet is captured, Zeek first conducts basic analysis throughout which events are generated, a script event handler has to be interpreted, and a call to the operating system has to be conducted, all of which introduce some time delay, resulting in time discrepancies in the order of hundredths of milliseconds possibly to low milliseconds.

- **network_time** - Choosing the network time for timestamping overcomes this problem. When a packet is first observed on a medium — online or offline — Zeek records the current time. Regardless of any processing that happens between the capture and `new_packet` event handling, this function returns the timestamp of the last processed packet by accessing the initially stored value. This means that the current time is always greater or equal to the network time in Zeek's environment.

The extraction of the rest of the extended flow information is more straightforward. Even though only TCP, UDP, and other higher levels directly provide a `bool` field stating whether

a packet is sent from the originator to the responder or the other way around, determining the packet direction is a simple task. The originator address available at `connection` can be compared with the source address from the packet's IP header and in case of a match, the direction value will be 1; otherwise, it will be -1. When accessing the transport layer payload size, first it has to be considered what protocols are of interest. When tracking connections, Zeek considers three protocols, TCP, UDP, and ICMP. Now, the ICMP protocol is not actually a transport protocol as it does not facilitate data transfer between hosts; rather, it is used to send error messages and operational information. For this reason, ICMP packets can be simply approached as having a zero payload size. In the case of TCP and UDP packets, data length (dl) and UDP length (ulen) are conveniently accessible in their respective header records. The last PSTATS information is TCP flags, which are relevant only when analyzing TCP packets. When it is clear that a TCP packet is being analyzed, the TCP flags are also readily available as a TCP header record field, `flags`.

At this point, it is also necessary to address `pstats` fields recording numbers of packets and bytes, as well as the connection's termination time. Although the `connection` record provides endpoint and flow statistics that might be relevant, there are some issues that prevent their reliable use.

Bytes and Packets The first issue is that the number of bytes is concerned with IP-level bytes rather than transport-level bytes, and primarily that both these values are set only if `use_conn_size_analyzer` parameter is set to true in a global environment. An option would be to access these numbers at a connection's `conn` record field, however, the documentation states that the same boolean has to be set for the information to be collected and additionally that the number of bytes might be inaccurate. Hence, gathering the endpoint information from the `endpoint` and `conn` records is unreliable and insufficient in the case of certain configurations.

Calculation of these fields, however, does not require too much additional effort. When a packet direction is determined in the `new_packet` handler, the appropriate packet number field can be simply incremented by one — source packet number when the direction is 1, destination when -1. Addressing the number of bytes sent, these fields can be incremented in a similar fashion since the transport-layer payload size and direction have already been determined.

Termination Time To address the connection's end time, although it could be determined from the PSTATS timestamps vector, the value might not always be accurate. When a connection is longer than the number of packets to be analyzed for PSTATS data, its actual termination happens after the last logged timestamp. Another option would be to leverage the connection's start time and duration available through the `connection` record; however, the issue is that although the start time is accurate, the duration does not cover trailing TCP packets like the final ACK messages. Due to this feature, if an end time was calculated by adding the connection duration to its start time while all packets were accounted for in the PSTATS timestamps vector, the last timestamp would appear to be outside the connection's boundaries. The start time can be hence read from the `connection` record upon its termination, but to record the connection termination time 100% accurately, the value has to be updated with every analyzed packet in a connection. By employing this approach, the extension will provide accurate data where packet timestamps vector and connection's time boundaries are synchronized.

To better picture the script extension design, Figure 2.2 demonstrates the flow of events from Zeek's start to the record population. Zeek starts by parsing the script — loading record redefinitions, event handlers... — and continues by generating events that will be handled as described in this section.

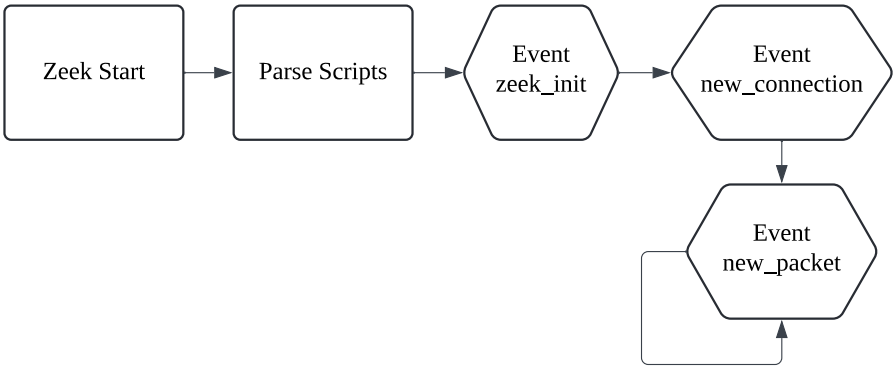

Figure 2.2 Script record population design

2.3.3.2 Compiled Plugin Design

Because of the high processing requirements for the extension, a stronger approach than a strictly script solution has to be employed. Zeek's auxiliary init-plugin script creates a compiled plugin template that serves as a solid starting point for individual packet inspection. It was already mentioned that most Zeek extensions are a combination of both scripts and compiled parts. The idea for the compiled record population is to leverage the structures and events already defined in the script solution and only substitute the `new_packet` event handler with a C++ solution. The main challenge lies in facilitating interaction between the plugin and the script structures.

While the plugin builds upon the designed script, the extension flow is slightly altered, utilizing a different packet analysis approach, as seen in Figure 2.3. All differences are described in detail throughout the rest of this section.

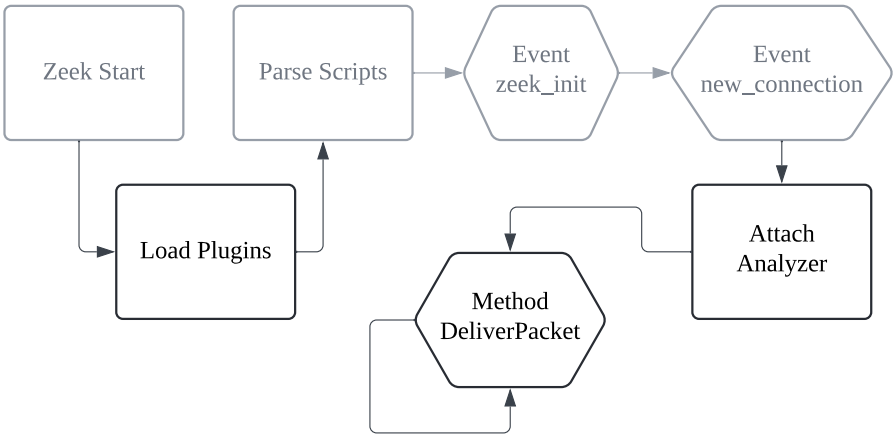

Figure 2.3 Plugin record population design

Component Selection First, it is important to explore the Zeek environment's components available for the purpose of packet analysis. Reaching out to the community at this point has been very helpful, as by consulting with Arne Welzel, one of the core Zeek developers, I was able to understand Zeek's analysis components, their interaction and management, and to adhere to best practices both through analysis and development of the extension.

I was first advised to utilize a general `Analyzer` class component. Compared to a lower-level packet analyzer component, a simple analyzer is additionally associated with a connection, which is very convenient for the purpose of flow data management. Packet analyzers operate below

Code listing 2.8 DeliverPacket method to be overridden for individual packet access

```
/**
 * Passes packet input to the analyzer for processing. The analyzer
 * will process the input with any support analyzers first and then
 * forward the data to DeliverPacket(), which derived classes can
 * override.
...
 */
void NextPacket(int len, const u_char* data, bool is_orig,
    uint64_t seq = -1, const IP_Hdr* ip = nullptr, int caplen = 0);

/**
 * Hook for accessing packet input for parsing. This is called by
 * NextPacket() and can be overridden by derived classes.
...
 */
virtual void DeliverPacket(int len, const u_char* data, bool orig,
    uint64_t seq, const IP_Hdr* ip, int caplen);
```

Zeek's session analysis on a link and network layer, parsing higher-level protocols and passing packets to appropriate protocol analyzers for session-level analysis, all of which is irrelevant to this extension.

For each connection, analyzers are kept in a tree structure where every analyzer has a parent analyzer and an arbitrary number of child analyzers. When an analyzer processes data, it forwards the rest further to all his children — for example IP analyzer might remove the IP header before passing the payload to TCP analyzer. The analysis conducted by this extension does not particularly require multiple levels of analysis, so defining one custom analyzer is sufficient to substitute the script's `new_packet` event handler functionality. As the extension needs to access transport layer packets before they are processed and the data is truncated, the idea is to assign the newly defined analyzer as a child of the tree root called session adapter.

Analyzer Design To accommodate the extracted data, the analyzer will have to define attributes for endpoint bytes and packets as unsigned integers, last packet time as `double`, and the four PSTATS vectors: vector of integers for directions (1 or -1), vector of `double` types for packet times, and two vectors of unsigned integers for payload lengths and TCP flags.

By exploring the capabilities and methods of the `Analyzer` class in Zeek's Github repository[12], I was able to grasp the class' API in the form of virtual methods. The most appropriate method that can be leveraged for session packets' access is a `DeliverPacket` method with the description and definition in Code Listing 2.8. This function specifically can conduct the packet-level data extraction similarly to the script's `new_packet` handler — as depicted in Figure 2.3.

- **Time** - First addressing the packet timestamping, where the initial approach was to leverage the C++ chrono library. This solution, however, introduces the same issue as the `current_time` function available in Zeek's script environment, so it is imperative that the compiled solution accesses the last packet timestamp as `network_time` does. I have been advised by Vern Paxson, the original Bro designer, that the same value is accessible in the C++ environment under `zeek::run_state::network_time`, which has saved me the effort of browsing through Zeek's source code. This accurate value can then be used for both the last packet time update and for timestamp vector assignment.
- **Direction Determination** - In the plugin solution, determining the packet direction is trivial, as the `DeliverPacket` function directly provides it in the form of a boolean parameter

Code listing 2.9 FTP analyzer component registration

```
class Plugin : public zeek::plugin::Plugin {
public:
    zeek::plugin::Configuration Configure() override {
        // Component registration
        AddComponent(new zeek::analyzer::Component("FTP",
            zeek::analyzer::ftp::FTP_Analyzer::Instantiate));
        AddComponent(new zeek::analyzer::Component("FTP_ADAT",
            nullptr));

        // Plugin configuration definition
        zeek::plugin::Configuration config;
        config.name = "Zeek::FTP";
        config.description = "FTP␣analyzer";
        return config;
    }
} plugin;
```

`orig`, stating whether the packet is being sent from originator to responder or not – meaning responder to the originator.

- **Payload and Flags** - To access the transport payload size and TCP flags, the extension has to process the `IP_Hdr` object being passed as an `ip` parameter. The `IP_Hdr` class is a class used for IPv4 and IPv6 packet header wrapping which can be leveraged for next protocol identification using its `NextProto` method. Zeek does not provide similar header wrappers for TCP, UDP, or ICMP protocols, so other networking C++ libraries must be leveraged for the next headers' parsing. Both the TCP and UDP payload sizes and TCP flags are parts of the transport protocol headers, so they do not have to be calculated but rather simply read from the headers, while ICMP payload size remains zero, as proposed in the script solution.
- **Endpoint Statistics** - To address the numbers of packets and transport-layer bytes sent, the extension will update the attributes in the same way as the script solution: increment the appropriate packet count by one and add extracted bytes to the endpoint attribute based on direction.

Component Registration and Integration When a new analyzer is defined, it is necessary first to register it as a component. This is important as when an analyzer is being instantiated, either a name or tag must be provided, both of which must match a corresponding registered component's attributes. In other cases, Zeek's analyzer manager does not allow the analyzer's instantiation. The plugin API provides three methods that are executed before Zeek actually starts to analyze traffic, hence each suitable for correct component registration: `InitPreScript`, `InitiPostScript`, and `Configure`. The first two methods are executed by Zeek before and after scripts are parsed, while `Configure` is called when "the plugin is instantiated to query basic configuration parameters" [12] and returns a configuration object describing the plugin. Although component registration would be acceptable in all three methods, after exploring other plugins in Zeek's repository, it was obvious that component registration is typically conducted in the `Configure` method before the configuration is defined and returned; hence this extension will follow the convention — during plugin load in Figure 2.3. An example of how the FTP plugin registers an analyzer component can be seen in Code Listing 2.9.

When the component is successfully registered in the system, it has to be assigned to each connection before the actual analysis is conducted. The plugin API provides several hook methods that act as entry points for the plugin to various events such as log initiation, network

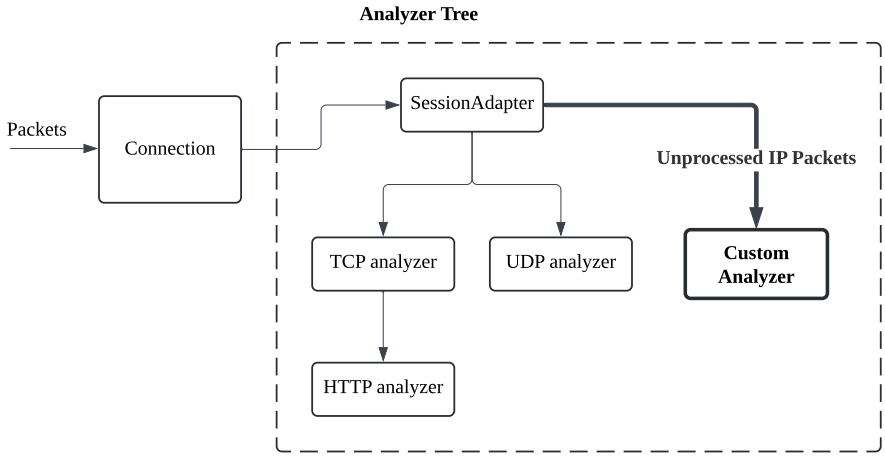

Figure 2.4 Custom analyzer attachment

time update, and, most importantly, the setup of connections' analyzer trees, which I was advised by Arne Welzel to utilize for the purpose of analyzer attachment to a connection. This hook "executes when a connection's initial analyzer tree has been fully set up" [12], at which moment the tree can be manipulated and extended. The new analyzer can then be assigned as a `ChildAnalyzer` of the tree root, facilitating the individual packet inspection as previously discussed. An example of a custom analyzer assignment to the connection's analyzer tree is presented in Figure 2.4.

Value Management When the analyzer collects the desired data, one of the main challenges the extension faces is that the data stored in a C++ plugin is not accessible in the script environment, essentially leaving `pstats` record fields empty. To make the data available, the `connection` record has to be accessed and updated throughout the analysis. Zeek's plugin API provides a method specifically designated for this purpose called `UpdateConnVal`, which is called whenever a `connection` record is accessed in the script for the information it holds to be accurate and up to date. Now, the method provides only a `RecordVal` pointer to a `connection` record which has to be further navigated to retrieve the custom record and update its appropriate fields. Since it is unclear in Zeek's event-driven architecture whether the `pstats` instantiation and assignment in the main plugin script's `new_connection` precedes the update call, it is important that the extension first checks whether `pstats` has been assigned and eventually creates and assigns it.

To create a Zeek script record in a C++ plugin, the record type describing the record structure — defined in the main Zeek script — has to be retrieved. After the record of the desired type is instantiated, it has to be correctly assigned to the `connection`. When a record type is defined, Zeek indexes all the fields it encapsulates, beginning with the number 0. When individual record fields need to be accessed, it is done so using the record's field indices. Rather than simply updating a value, a new Zeek value, such as an `int` or `count`, has to be first constructed and only then assigned to an offset as a whole. By implementing the connection update function, the compiled plugin solution completely substitutes the `new_packet` event handler functionality of populating the `pstats` record.

2.3.4 Connection Termination

With the packet-level data collection process established for both script and hybrid (script + compiled plugin) solutions, the next step is to finalize the collection of the remaining basic

flow data upon the connection's termination. The first considered event for this purpose was `connection_finished` based on its descriptive name, however, this event is generated only for TCP connections finished by a FIN message from both endpoints, which would again result in an incomplete analysis, as it would omit data from UDP and ICMP flows. A more suitable event is a `connection_state_remove` event which is "generated when a connection's internal state is about to be removed from memory"[3]. By choosing this event, termination of all types of connections is covered, and all required data can be extracted.

When the `pstats` record was instantiated, both source and destination IP addresses and port numbers were already available and recorded. Then, both script and hybrid solutions populated the PSTATS vectors with directions, transport layer payload sizes, packet timestamps, and TCP flags while updating the byte count, packet count, and last packet time fields. The only remaining fields to be addressed are hence MAC addresses, connection initiation time, and transport protocol.

Beginning with MAC addresses, the extension can leverage the `endpoint` records available as `connection` record fields, which directly provide link-layer addresses as `l2_addr`. The transport protocol is also provided by Zeek as `transport_proto` enum field in connection's `conn` record field. The only issue is that protocols are described by their abbreviations – "TCP", "udp", and "icmp" – which means they have to be converted to their IP numerical identifiers – 6, 17, and 1, respectively. Other protocols should not be encountered as Zeek only tracks the 3 addressed protocols, however, as the enum contains also an `unknown_transport` value, it will default to zero in that case.

2.3.5 Extension Parameters

For a more granular control over Zeek's configuration, script values can be defined from the command line when a Zeek instance is being launched. This extension requires the definition of 3 parameters to facilitate control over its functionality: `LogFirst`, `LogEmpty`, and `OutIfcSpec`. To enable setting these parameters during startup, they have to be globally accessible, which can be achieved by declaring them in the export block next to the definition of `Pstats::Info` record. When variables are exported as a part of a namespace, their value can be specified as shown in Code Listing 2.10.

Code listing 2.10 Zeek launch command specifying unix socket output interface

```
$ zeek -i any scripts/pstats Pstats::OutInterface="u:1111"
```

As the output interface specification is necessary for the extension to function as intended, this parameter must be mandatory and proper checks must be implemented to ensure a specifier was provided. The remaining parameters are also crucial for the extension to effectively work with data — especially very long connections in terms of packets — these parameters can be set to a sound default value and then optionally redefined if necessary. The NEMEA's PSTATS module gathers statistics for the first 30 packets in a connection by default and does not log zero transport-layer payload packets, hence this extension sets the parameters to the same values. To actually leverage the parameters in the extension, their use has to be implemented properly in both script and hybrid solutions. The following sections discuss the integration of `LogFirst` and `LogEmpty` parameters, while `OutIfcSpec` is addressed in the subsequent Logging and Export Section 2.3.6.

2.3.5.1 Script Solution

As the variables are defined directly in the Zeek script, accessing them throughout the script solution is straightforward. Now as both `LogFirst` and `LogEmpty` affect whether packets are

Code listing 2.11 Packet analysis pseudocode

```
# Base Functionality
- Get timestamp
- Update time of connection's end
- Extract payload size
- Determine direction
- Increment endpoint packets
- Add payload size to endpoint bytes

# Extended Data
- Check whether packet-level data should be recorded
  and eventually end this packet's analysis
    - Either if packet is empty and LogEmpty is set to false
    - Or maximum packet threshold has been reached

- Extract TCP flags
- Update PSTATS vectors
```

analyzed for extended flow data, the extension addresses both values in the individual packet inspection facilitated by `new_packet` event handler. Since both parameters are related only to extended PSTATS data, the basic analysis has to remain unaffected by their integration.

The base analysis conducted, regardless of whether a maximum analyzed packet threshold has been reached or an empty packet was encountered, includes firstly updating the connection end timestamp, determining packet direction for packet count incrementation, and extraction of payload size for it to be added to the appropriate endpoint bytes. This means that all information to be added to the PSTATS vectors except TCP flags is available without any extra effort. Based on this observation, the only functionality that can be positioned after the parameters' check is the TCP flag extraction and the PSTATS vectors' update itself. A pseudocode describing the `new_packet` handler flow can be seen in Code Listing 2.11.

2.3.5.2 Hybrid Solution

The challenge faced when utilizing a compiled packet analysis again lies in the value management between the Zeek script and the C++ plugin. Once the global variables' values are successfully loaded into the C++ environment, their use is very similar to the script's `new_packet` event handler.

Compared to the `pstats` values assigned to the `connection` record, the `LoadFirst` and `LoadEmpty` values are not attached to any record but are rather available in the module's namespace. This means they have to be loaded in a different way than by leveraging record offsets. Zeek refers to script object names as IDs or identifiers and provides a method `zeek::id::find` specifically designated for the purpose of global variables' retrieval using their name, such as `Pstats::LogFirst`.

To manage these values effectively, their loading has to be correctly positioned in the extension's architecture. If both values were being looked up in the analyzer's `DeliverPacket` method, the script would again be referenced with every incoming packet, which introduces redundancy as these values remain static from Zeek's start until its termination. Loading the values during the initiation of the custom analyzer would also be redundant as the new analyzer is instantiated for every new connection, which could even result in the same overhead as the previous case in case of very short connections — analyzers being frequently generated. This leads to the idea that the values should be loaded only once per Zeek instance's lifetime, ideally during the plugin's initiation phases. Because the variables are defined in a script, it is necessary

to look up the values only after the script has been parsed and command line parameters are correctly assigned. Plugin's `InitPostScript` virtual method presents a perfect entry point as it is concerned with "second-stage initialization of the plugin called late during Zeek's startup after scripts are parsed" [12]. The values can be conveniently stored as the plugin's attributes and accessed by analyzers without the need to look them up in the script environment.

2.3.6 Logging and Export

It has already been pointed out that Zeek's reporting is facilitated by default through multiple ASCII logs, where each log reports a different protocol or other analysis results. To follow this convention, all exported data by this extension will be first logged in a designated TSV log pstats.log for complete transparency on the exporter side of the flow monitoring architecture. A UniRec message will be constructed and sent to the NEMEA flow collector before the local TSV logging to minimize the delay between collection on Zeek's side and analysis in NEMEA. Since UniRec is a binary format and there is no existing solution for its construction in Zeek script, the export implementation is expected to be much simpler in a C++ environment where existing NEMEA libraries can be leveraged.

Traditional logging of custom data is a common task for Zeek extensions, and the implementation process is well documented by Zeek [3]. The Zeek logging framework abstracts much of the process from file creation to appending records. Since the information collection is completed in the `connection_state_remove` handler in both script and hybrid solutions, and logging is commonly defined using the Zeek script, the local logging design does not differ between the solutions.

Custom logging design begins with a definition of a log stream, which corresponds to a single log output. To inform Zeek about a new log stream associated with the extension, a `Log::ID` enumerable holding all Zeek instance's log streams has to be redefined to include a `Pstats::LOG` additionally. Since the changes have to be globally visible, this redefinition has to be conducted in an export block.

Before writing data to a stream, it is necessary to specify the logged data format during its initiation, for which the already defined `Pstats::Info` record can be used — with only a minor addition of `&log` attribute to fields that are to be logged, in this case, all fields. To follow Zeek's guidelines, the creation of a log stream is conducted in `zeek_init` event handler, which is executed during Zeek's initialization before any processing begins. Once a stream is initiated, `pstats` data can be written to it using a `Log::write` method by providing an appropriate log identifier, resulting in log file creation and its population.

Addressing data export over the network is a much more complicated task, which requires a complex solution in both script and hybrid options.

2.3.6.1 Script Export

Although Zeek script is a Turing-complete language, facilitating network communication and binary structure construction and manipulation is not supported by default and its implementation would go beyond practical limits. To satisfy the requirements, supporting programs or scripts would have to be introduced, and their integration with Zeek would have to be facilitated using Zeek script functions `system` or `piped_exec`, which enable scripts to communicate with the underlying operating system and other processes. For this reason, this work does not discuss the script UniRec export further.

2.3.6.2 Hybrid Export

When enhancing a script with compiled components, it is important to decide how to achieve the export of collected data. Since most of the connection data is available in each analyzer — fields

Code listing 2.12 Conn extension BiF definition

```
function set_current_conn_duration_threshold%(cid: conn_id,
        threshold: interval%): bool
%{
zeek::analyzer::Analyzer* a = GetConnsizeAnalyzer(cid);
if ( ! a )
    return zeek::val_mgr->False();

static_cast<zeek::analyzer::conn_size::ConnSize_Analyzer*>(a)
        ->SetDurationThreshold(threshold);

return zeek::val_mgr->True();
%}
```

such as protocol and MAC addresses would have to be additionally read in the C++ code — it would be intuitive to construct and export UniRec messages in the analyzer's `Done` method, which is invoked when a connection has been completely analyzed. Although this decision would result in better performance by avoiding further interaction with the script, the extension will adhere to Zeek's best practices and implement both export and logging by updating the script-defined record and the `connection` object.

This leads to two options: implement a specialized writer component and register it in the system or create a BiF compiled function which would be called in `connection_state_remove` before ASCII logging of each connection. I was advised by the Zeek community to keep things simple and opt for the BiF function, where the implementation does not need to leverage Zeek's component API further.

Extraction Function The generated plugin skeleton contains a pstats.bif file, where custom script functions and events can be defined. This file will hence be used to define a compiled function, which will accept the collected flow data and take care of its export to NEMEA. To grasp the bif file environment and syntax, I inspected other extensions defining their custom functions, such as Conn extension Code Listing 2.12.

Similarly to updating script structures from C++ code in Section 2.3.3.2, the structure values now have to be read back to the plugin environment to leverage NEMEA's C++ libraries for packing and export. The processing function will hence start by accessing all `pstats` record's fields, including vectors, and storing them in appropriate type local variables.

UniRec Interface Leveraging the NEMEA framework, the extension will make use of its unirec++ library, which provides structures and methods designed for managing low-level TRAP interfaces and UniRec message construction. By consulting with NEMEA developer Pavel Šiška and by observing a C++ NEMEA module torder, I have been able to identify necessary steps for both interface setup and UniRec record construction and population.

A key class, defined by the unirec++ library, is `Nemea::Unirec`. An object of this class encapsulates the NEMEA module's TRAP interfaces and abstracts their low-level configuration. After an object is created, it is imperative that it is kept alive for the whole Zeek extension execution, hence the extension will instantiate it in the plugin's configuration phase and store it as an attribute of the plugin class. This way the extension opens an output interface only once, avoiding redundancy.

The decision of where exactly to initiate the interface in the configuration phase requires a careful approach. As the output interface specifier is required for the `Unirec` initiation, it is necessary that the main.zeek script has already been loaded. A convenient approach would be to leverage the plugin's `InitPostScript` method similarly to the extension parameter setup

Code listing 2.13 UniRec record template design

```
ipaddr   DST_IP,
ipaddr   SRC_IP,
uint16   DST_PORT,
uint16   SRC_PORT,
uint64   BYTES,
uint64   BYTES_REV,
time     TIME_FIRST,
time     TIME_LAST,
macaddr  DST_MAC,
macaddr  SRC_MAC,
uint32   PACKETS,
uint32   PACKETS_REV,
uint8    PROTOCOL,
int8*    PPI_PKT_DIRECTIONS,
uint32*  PPI_PKT_LENGTHS,
time*    PPI_PKT_TIMES,
uint8*   PPI_PKT_FLAGS
```

in Section 2.3.5.2; however, an issue arises when active Zeek plugins are listed using `zeek -N`. For the plugins to be listed completely, including their registered hooks and components, the `InitPostScript` is executed for each, and since unsuccessful interface initiation is designed to exit the program, such a listing would not be possible. For this reason, the extension will employ a simple workaround by defining another BiF function for UniRec interface configuration, which will be called in the main script's `zeek_init` handler that only executes when Zeek is launched — Zeek init event in Figure 2.5.

To instantiate a `Nemea::Unirec` object, module info has to be provided, specifying the number of input and output interfaces. As the network data input is taken care of by Zeek either from a file or network interface, the extension will only define an output interface specified by the extension's parameter `OutIfcSpec` — a string in a correct TRAP interface specifier format [13].

If a specified interface is available, the `Unirec` object method `buildOutputInterface` can be used to build a `UnirecOutputInterface` class object. Similarly to the main `Unirec` object, the interface will be kept within the plugin class for convenient access throughout the extension's lifetime.

Template Design The next step is to define a UniRec message template describing individual record fields and associate it with the interface. The NEMEA framework defines UniRec field types similar to C++ and Zeek, hence the template design closely resembles the Zeek `Pstats::Info` record type. The proposed template in Code Listing 2.13 has been designed according to NEMEA field standard [14] in terms of both field names and their data types, ensuring compatibility and efficiency.

Once the template is defined and associated with the output interface, the interface object has a method `getUnirecRecord` which "gets a reference to the pre-allocated UniRec record for efficient use. This function provides access to the UniRec record instance that has already been pre-allocated within the `UnirecOutputInterface`. It allows direct modification of the record's fields before sending it through the TRAP interface" [14]. As the function comment states, using the record reference, its individual fields can be updated with collected values. As opposed to the interface setup which suffices only once per extension's lifetime, the record field value assignment has to be conducted with every terminated connection, hence takes place right after data is read from script structures in the final export BiF function. The assignment is very similar to the Zeek record update, leveraging individual field offsets in the UniRec record.

At this point, the UniRec record is finalized and can be sent to the output interface using its `send` method. Collectors can then simply connect to the exporter — locally or over the network — and subscribe for collection, or in case a file output is selected, data can be observed in a specified output file. To correctly terminate the extension and its export, UniRec objects have to be correctly destructed so that all buffered messages are flushed to the interface, avoiding data loss.

2.3.7 Solution Comparison

When comparing the two suggested solutions throughout all extension phases, it is immediately evident that the design of the script-only solution is much less complicated as opposed to the compiled enhancement. While the script's individual packet is directly facilitated by the `new_packet` event, a whole new compiled plugin structure must be introduced before any functionality is implemented. To achieve the packet access itself, a new analyzer component has to be defined, registered, and correctly assigned to each connection manually. Furthermore, to substitute the `new_packet` event handler functionality completely and have the collected data by an analyzer available in the Zeek script structures, the compiled plugin has to update the appropriate structures whenever they are to be accessed in the script environment. All of these tasks require an advanced knowledge of both C++ programming language and Zeek's C++ API. As both Zeek and NEMEA are open-source tools, they often miss crucial documentation necessary to fully grasp their components and functions, which led me to reach out to the open-source community for guidance.

On the other hand, due to the shortcomings of the Zeek script in performance and UniRec extraction, an extension must employ a more effective solution than solely Zeek's script. Compiled code runs at native speed, which significantly reduces the overhead compared to interpreted scripts. Although the plugin still leverages the Zeek script for structure definition and connection-level events, the script is being interpreted only a few times per connection, completely avoiding the packet-level event handler and its overhead.

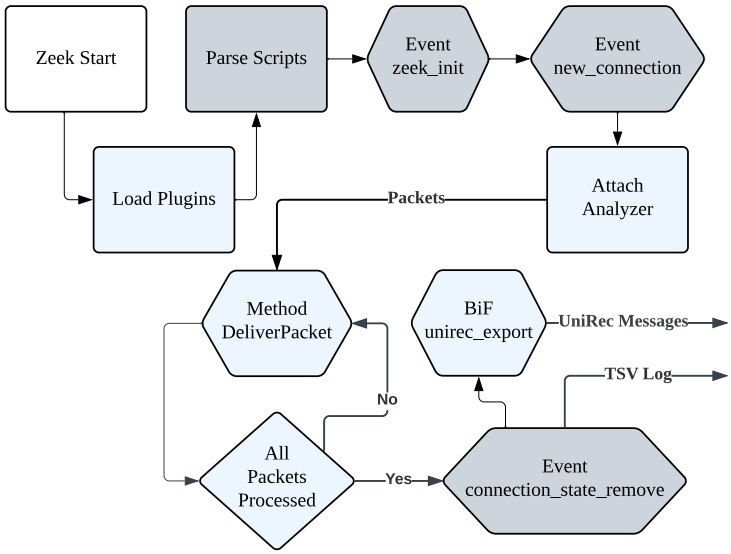

Figure 2.5 Complete plugin design

In conclusion, while Zeek's script can be easily and rapidly deployed in lower processing environments or for testing, more complex production-level extensions require some compiled enhancement. Despite the improvements the compiled solution presents, in some cases – like

a high number of one-packet connections forcing frequent script execution – the performance difference between the solutions might be less substantial.

The complete design of the plugin solution is presented in Figure 2.5, where light-blue objects are plugin-related and gray script-related. After Zeek is started, components, hooks, and BiF elements are registered when plugins are loaded. With every new connection, the registered analyzer is attached and collects flow statistics from all connection's packets. Finally, when all packets are processed, the control is passed to the connection termination event handler which logs the collected data in a TSV log. It also invokes the UniRec export function which exports all data to the UniRec output interface.

Chapter 3
Implementation

This chapter details the practical implementation of the Zeek extension which was designed to collect and export extended flow statistics effectively. The implementation is divided into two main parts: the script-based solution and the hybrid solution extended with compiled parts.

The script-based solution, discussed in the first section, focuses on the definition of new structures, their population, and logging. Due to infeasible complexity, it does not implement export to NEMEA, which is one of the key requirements for the extension. However, since both solutions overlap, it serves as a foundation for the plugin solution.

The compiled plugin builds upon the script solution, defining and registering new components and custom functions, which is necessary both to achieve better performance and to utilize C++ libraries for a more convenient export of flow statistics.

3.1 Script

Implementation of the script solution was conducted in a single file pstats.zeek, beginning with module declaration `module Pstats;`, which defines the module namespace. Structure and global definitions are then conducted for the extension to correctly function in Zeek's environment. Finally, all event handlers are defined to conduct the proposed analysis, data collection, and logging.

3.1.1 Definitions

The initial module declaration is followed by an export block seen in Code Listing 3.1, which encapsulates all definitions and redefinitions according to the proposed design. The export begins with a redefinition of the global enum object containing all log identifiers, where `LOG` is, in essence, `Pstats::LOG`. It continues by definition of parameters that can be redefined from the command line during startup, initiated with discussed default values. The block ends with the custom `Pstats::Info` record, where all fields are marked for logging using the `&log` attribute. Attributes whose values are either not known at the time of record instantiation or cannot be set to a sound default value are further defined as `&optional` so that they can be assigned later during the analysis process. The `connection` record is then redefined to optionally include the newly defined record as specified in the record design Section 2.3.1.

Code listing 3.1 Custom Zeek record definition

```
export {
    # Add "Pstats::LOG" id to global list of log identifiers
    redef enum Log::ID += { LOG };

    # Specify these parameters at launch to modify script behavior
    global LogFirst: count  = 30    &redef;
    global LogEmpty: bool   = F     &redef;

    type Info: record {
        dst_ip:          addr                &log;
        src_ip:          addr                &log;
        dst_port:        port                &log;
        src_port:        port                &log;
        protocol:        count   &optional   &log;
        bytes:           count               &log;
        bytes_rev:       count               &log;
        time_first:      time    &optional   &log;
        time_last:       time    &optional   &log;
        dst_mac:         string  &optional   &log;
        src_mac:         string  &optional   &log;
        packets:         count               &log;
        packets_rev:     count               &log;
        pkt_directions:  vector of int       &log;
        pkt_lengths:     vector of count     &log;
        pkt_times:       vector of time      &log;
        pkt_flags:       vector of count     &log;
    };
}
```

Code listing 3.2 Zeek initiation event handler defining a log stream

```
event zeek_init () {
    Log::create_stream(Pstats::LOG, [$columns=Info, $path=}pstats}]);
}
```

3.1.2 Event Handlers

The first event handler defined by the script is the `zeek_init` event shown in Code Listing 3.2, which takes care of log stream creation, providing the `Pstats::Info` record as a template and "pstats" as a log file name, resulting in pstats.log file creation and its subsequent population.

Now that the environment and components are configured, the script defines event handlers focused on the actual data collection and logging.

New Connection Starting with record instantiation and assignment to the connection record, the `new_connection` event handler instantiates the `pstats` record with known IP addresses and ports and default values for non-optional fields. Endpoint bytes and packets are set to zero and later incremented, while vectors are set to empty vectors, ready to be populated with per-packet information. To access record members, Zeek makes use of dollar notation instead of dots compared to C or C++ to avoid confusion with IP addresses. The syntax can be demonstrated in both access to endpoint information in instantiation and assignment to `connection` in Code Listing 3.3. To test whether an optional field is present, a question mark can be placed before

the dollar sign, returning T (true) or F (false).

Code listing 3.3 Custom record instantiation and assignment

```
event new_connection (c:connection) {
    c$pstats = Info($dst_ip=c$id$resp_h,
                    $src_ip=c$id$orig_h,
                    $dst_port=c$id$resp_p,
                    $src_port=c$id$orig_p,
                    $dst_bytes=0,
                    $src_bytes=0,
                    $dst_packets=0,
                    $src_packets=0,
                    $pkt_directions=vector(),
                    $pkt_lens=vector(),
                    $pkt_flags=vector(),
                    $pkt_times=vector());
}
```

New Packet After successful instantiation of the `pstats` record, it can be accessed and updated in the `new_packet` event handler with every incoming packet. Starting with timestamping, the handler leverages the `network_time` function described in the script design to retrieve an accurate packet timestamp. The connection's end time can then be immediately updated.

To minimize the number of conditional statements, the next addressed information is packet bytes using the designed approach of extracting payload sizes from protocol header structures, which is simply set to zero in case of ICMP or unknown protocols. Next, when packet direction is determined by address comparison, both packet and byte count can be updated in one place.

At this point, further analysis and assignments are conducted only in case the following condition in Code Listing 3.4 is not met, meaning empty packets are not logged when they are not supposed to, and no additional packets are logged when a provided or default threshold has been reached. If the condition is not met, the `new_packet` event handler finishes by updating the record vectors with direction, bytes, timestamp, and if applicable — `if (p?$tcp)` — also TCP flags using a `+=` vector assignment operator.

Connection Termination The last event handler defined in the script solution handles the `connection_state_remove` event. Based on the proposed design, all remaining fields are assigned before a record is logged. Starting with the MAC addresses, the presence of `l2_addr` fields is tested using the question mark notation, which is eventually assigned to the `pstats` record fields `dst_mac` and `src_mac`. Next, a Zeek switch statement very similar to C or C++ is leveraged for protocol translation from `enum` type to standard IP header protocol field numbers. Before comparison of the `enum` values with literal strings such as "tcp", a conversion from `enum` to string has to be conducted using `cat(enum)` function, as can be seen in Code Listing 3.5.

To finalize the information collection, the event handler retrieves a mandatory `connection` record's field `start_time` and assigns it to `pstats$time_first` field. Now, the populated

Code listing 3.4 Extension parameter check during analysis

```
if ((LogFirst < c$pstats$dst_packets + c$pstats$src_packets)
        || (LogEmpty == F && bytes == 0))
    return;
```

Code listing 3.5 Protocol abbreviation translation

```
# Translate protocol string to corresponding
# IP header protocol field number
if (c?$conn && c$conn?$proto) {
    local proto = cat(c$conn$proto);
    switch proto {
        case "tcp":
            c$pstats$proto = 6;
            break;
        case "udp":
            c$pstats$proto = 17;
            break;
        case "icmp":
            c$pstats$proto = 1;
            break;
    }
}
```

custom record can be logged by providing the defined log identifier in the following fashion: `Log::write(Pstats::LOG, c$pstats);`.

It was decided that implementing export of flow records in UniRec messages leveraging Zeek's scripting language would be too complex and inconvenient, hence this is the final form of the script-based solution.

3.2 Plugin

To satisfy the UniRec export requirement, a compiled solution has to be implemented for performance, convenience, and security – using Zeek functions like `system(command)` when unnecessary is not considered secure. Although this solution largely builds upon the implemented script, adding new functionality in C++ involves many challenges, from leveraging Zeek API to integration with foreign NEMEA libraries.

To initiate the plugin structure, Zeek's init-plugin script was leveraged, creating a "Pstats" plugin in "Nemea" namespace. The generated directory is then populated with the main Zeek script and C++ source and header files, resulting in an installable Zeek plugin. For the main script to be loaded with the plugin, it is placed in `scripts/Nemea/Pstats` directory and loaded by `scripts/Nemea/Pstats/__load__.zeek`, which is further loaded by `scripts/__load__.zeek`. Although the structure might seem complicated, it presents a scalable approach to possible future enhancements.

3.2.1 Main Script

Adhering to the proposed design, the compiled plugin interacts with a main.zeek script, which still facilitates new record definition, its instantiation, data collection finalization, and ASCII logging — largely based on pstats.zeek script solution. The main difference is that the `new_packet` event handler is completely omitted as the individual packet analysis is compiled. Although vector and counter fields of the defined `Pstats::Info` record are being assigned in the plugin, it is considered a bad practice to leave counter variables uninitialized, hence the record instantiation `new_connection` will remain untouched.

Additionally, as this solution implements data export, an `OutIfcSpec` parameter specifying the TRAP output interface is exported as a global variable for redefinition upon Zeek instance

startup, initially set to value "undefined". The last addition to the main script is two new functions: `unirec_export`, which exports data before ASCII logging, and `unirec_init`, responsible for UniRec output interface initiation. Both of these functions are described in depth in the following sections.

3.2.2 BiF

Similarly to other extensions compiling new Zeek functions, the plugin starts with function declaration in pstats.bif file. For convenient development — Visual Studio Code, in which the plugin was developed, does not provide an extension that could handle the bif file formatting — the file only defines function wrappers which then call imported plugin methods `UnirecInit` and `UnirecExport`.

In the declaration in Code Listing 3.6, the header files are first included in a special C block, following the syntax observed in other extensions. As Zeek cannot define functions without a return value, the wrappers are defined to return a boolean symbolizing either success or failure. The actual functionality conducted by the imported functions is discussed in Section 3.2.3 and Section 3.2.5, respectively.

Code listing 3.6 BiF function wrappers

```
%%{
#include "zeek/Val.h"
#include "Plugin.hpp"
%%}

function unirec_init%(%): bool
    %{
    if (::plugin::Nemea_Pstats::plugin.UnirecInit())
        return zeek::val_mgr->True();
    return zeek::val_mgr->False();
    %}

# Zeek wrapper function for UniRec message construction and export
function unirec_export%(c:connection%): bool
    %{
    // Call the actual export function defined in Export.cc
    if (::plugin::Nemea_Pstats::plugin.UnirecExport(c))
        return zeek::val_mgr->True();
    return zeek::val_mgr->False();
    %}
```

3.2.3 Plugin Class

The implementation of the compiled plugin begins with the definition of a `Plugin` class, which inherits from the `zeek::plugin::Plugin` class, facilitating the integration with Zeek's core. For better organization, the class and method definitions and declarations are split into two files, Plugin.cc and Plugin.hpp. The base class provides multiple virtual methods that can be overridden and later invoked by Zeek in certain situations. Although the only required method for the plugin to be considered a valid Zeek plugin is a `Configure` method, other virtual methods are overridden, and new methods are defined for plugin initialization, component attachment, export, and proper cleanup. As the UniRec export is complex and requires separate source files for better organization, it is described in depth in a dedicated Section 3.2.5.

3.2.3.1 Configuration

Starting with the configuration, the method first enables a hook necessary for the plugin's `HookSetupAnalyzerTree` method to be invoked by Zeek, a necessary entry point for analyzer component attachment to a connection. The function takes a `HOOK_SETUP_ANALYZER_TREE` hook type enum and a priority value, which can be set to manage the order of this method invocation across all plugins. Since this plugin does not rely on others, the value is set to zero.

Next, the configuration method registers the analyzer component `Pstats_Analyzer` whose implementation is discussed in the next section. The registration is achieved using Zeek's `AddComponent` function, which registers the provided component. To construct such a component, a component identification name is provided with a pointer to its `Instantiate` method which returns a new object of the class. After that, the configuration method fulfills its purpose by creating a `zeek::plugin::Configuration` object and defining its attributes – plugin name `Nemea::Pstats` already prepared by plugin-init, a brief description, and version specification 1.0.0 – before returned.

3.2.3.2 Analyzer Attachment

Delving into the analyzer tree setup, the biggest challenge is to properly attach a new analyzer to the `zeek::Connection* conn` provided as a parameter to the hook function. The first step is to retrieve the connection's session adapter – essentially connection analyzer tree root – by invoking `conn->GetSessionAdapter()`. After the instantiation of a new analyzer by the means of Zeek's analyzer manager, an object that maintains and schedules available protocol analyzers, a segmentation fault kept arising during its attachment. Thanks to Arne Welzel's help, I have been able to solve this issue by treating attachments differently in TCP connections. The issue is that although analyzers are normally attached to session adapter using its `AddChildAnalyzer` method, TCP connections' session adapter is not actually an `IP::SessionAdapter`, but rather a `TCP::TCPSessionAdapter`, so it cannot be treated in the same way. This complication was solved by statically casting the adapter to a TCP adapter in case of TCP connections. After the attachment shown in Code Listing 3.7, the analyzer is prepared to process incoming traffic.

Code listing 3.7 Custom analyzer attachment

```
if (conn->ConnTransport() == TRANSPORT_TCP) {
    auto* tcp_session_adapter =
        static_cast<zeek::packet_analysis::
        TCP::TCPSessionAdapter*>(session_adapter);

    tcp_session_adapter->AddChildPacketAnalyzer(pstats_analyzer);
} else
    session_adapter->AddChildAnalyzer(pstats_analyzer);
```

3.2.3.3 Parameters

Another overridden virtual method leveraged during the plugin's initiation is an `InitPostScript` method that reads values of the script parameters: `LogFirst`, `LogEmpty`, and `OutIfcSpec`. For storage of these values, 3 corresponding class attributes are defined. Considering the maximum possible number of packets defined by the UniRec field standard, the first parameter is stored as a `uint32_t` type variable for effective memory management.

For each of the three parameters, a lookup is conducted using their name in the script's `Pstats::` namespace. When a pointer is retrieved, the function checks whether there is an associated value of the expected Zeek type and eventually reads the value and assigns it to the plugin

member variable. If the retrieval fails, the values are set to default values defined in the header file — `LOG_FIRST_DEFAULT 30`, `LOG_EMPTY_DEFAULT false`, and `OUT_IFC_DEFAUTL "undefined"`.

3.2.3.4 UniRec Interface

The UniRec interface setup method `UnirecInit` invoked by the `unirec_init` BiF function initiates an output interface by leveraging NEMEA framework libraries. As the UniRec export is one of the base extension requirements, it must be correctly initiated. The method hence first checks whether the `m_out_ifc_spec` member value was changed from the default value and eventually terminates the execution using the `exit` function with return code 1 while providing hints to users on how to specify parameters. If the check is passed, a `Nemea::Unirec` object is created by providing trap module information — 0 input interfaces, 1 output interface, module name, and description — and stored as a plugin class member for its persistence throughout the Zeek instance's lifetime. The subsequent interface initiation is conducted in a try-catch block as unsuccessful initiation raises exceptions.

As NEMEA modules are typically started from a command line, the initiation parameters are parsed from `argc` and `argv` parameters. To provide these parameters, both are created synthetically by constructing a `nullptr`-terminated C string array and simply calculating the arguments by eye, as this number won't change — Code Listing 3.8.

Code listing 3.8 Command line parameter synthetic creation

```
const char* argv[] = {}ProgramName}, "-i",
    m_out_ifc_spec.c_str(), nullptr};
int argc = 3;
```

These two values are then passed to `m_unirec->init(argc, (char**)argv)` method, which initiates the `Unirec` object, enabling a subsequent interface build. First, however, the plugin method checks whether an output interface is available and eventually notifies the user while exiting the program with return code 2. A `buildOutputInterface` method is used for the construction of `Nemea::UnirecOutputInterface` object that is also assigned as a plugin member for convenient access throughout the extension, specifically during the export phase. The last step is to change the UniRec record template of the interface to the one defined in design section 2.3.6.2 — defined in the plugin header file. In case an exception occurs during interface initiation, the method prints the exception and provides a reference to correct interface specification practices.

To demonstrate the challenges faced while navigating open-source software, whose development is often an ongoing process, a caught exception generated by NEMEA's unirec++ library using `std::cerr << "EXCEPTION: " << ex.what()` is included in Code Listing 3.9. This situation led me to explore the NEMEA framework's Github source code, and I have been able to identify the issue statically.

The `UnirecInit` method completes the main plugin class — except for the export discussed later — and finalizes the UniRec record export environment preparation.

Code listing 3.9 Exception generated by NEMEA framework

```
$ zeek -r pcaps/smallFlows.pcap
EXCEPTION: TODO
```

3.2.4 Analyzer

To adhere to best coding practices and enhance readability, the `Pstats_Analyzer` class, which inherits from `zeek::analyzer::Analyzer`, is also divided into header and source files Analyzer.hpp and Analyzer.cc. The base class also represents Zeek's API by declaring virtual methods to be overridden by developers to introduce custom functionality.

To facilitate the analyzer component registration in the plugin's configuration described in Section 3.2.3.1, a constructor and the instantiation method have to be defined. Following other extension conventions, the constructor is declared as explicit to prevent implicit conversions from its `zeek::Connection` parameter. The static `Instantiate` method then simply allocates and returns a new `Pstats_Analyzer` object and can be readily used in the `Component` instantiation process.

3.2.4.1 DeliverPacket

To substitute the script's `new_packet` event handler, the class overrides a `DeliverPacket` method that conducts the packet-level analysis. Before data is actually extracted, the analyzer class defines member variables as proposed in the analyzer design, addressing endpoints' byte and packet count, the time of the last packet that has to be constantly updated, and four PSTATS vectors. The primitive type members are set to zero in the class constructor, while vectors are preallocated using the `reserve` method to the `LogFirst` parameter value to avoid time-consuming reallocation during analysis and enhance performance.

The implementation of packet analysis is conceptually similar to the `new_packet` event with only minor changes, such as invoking a base class method to maintain functionality. The accurate timestamp is then read from `run_state::network_time` directly to member variable `m_time_last` instead of the local variable to optimize memory management. Another difference compared to the script approach is that TCP flags are always extracted regardless of whether they will be logged to avoid duplicate parsing of the header — it has to be always accessed to determine the payload bytes. The order in which protocols are checked begins with TCP protocol as it is encountered more commonly than UDP, which also enhances performance by minimizing conditional jumps during execution. In the case of ICMP or any other unknown protocol, the payload size remains zero.

Same as in the script solution, a check is conducted to verify whether PSTATS information should be logged — empty packets and maximum packet count. If not, the method simply returns, and the analyzer continues with the delivery of subsequent packets. In a heavy-load network, this method will be invoked very frequently; therefore, all functionality has to be optimized to the greatest extent possible.

3.2.4.2 UpdateConnVal

When all required connection information is collected, there is a need to update the script `connection$pstats` fields for subsequent TSV logging. Another overridden API method is hence `UpdateConnVal`, invoked by Zeek whenever `connection` is referenced in the script. The method provides a pointer to the `connection` record, which is immediately leveraged to retrieve a pointer to the `pstats` record. The function then checks whether retrieval of the `connection$pstats` record pointer was successful and eventually creates and assigns the record.

To increase readability, the code is modularized by introduction of a helper boolean function `CreateAssignPstats`, defined to create and assign the record. First, a Zeek record type has to be retrieved using `auto info_type = id::find_type<RecordType>("Pstats::Info")`. Then, a new `pstats` record is created using Zeek's `make_intrusive` function, which creates a reference counted object. Finally, using `pstats` field offset in the `connection` record's fields, the record is assigned. Errors encountered in all of these operations are handled by the false value returned and

eventually accompanied by an error message under the condition `DEBUG` was defined during compilation — wrapped in an `#ifdef-#endif` block. If the function fails, `UpdateConnVal` execution is terminated, and the assignment is conducted either in the script or the next `UpdateConnVal` call.

Once the field is retrieved or correctly assigned, the method can continue with the designed update. Helper functions `UpdateCount`, `UpdateTime`, and templated `UpdateVector` function are defined to enhance readability and to avoid code duplication. The count update function is used for the update of endpoint bytes and packets, while the time update function sets the time of the last connection's packet. Both functions start by finding the provided field offset using its name but differ in how the actual Zeek value is created before the assignment. Zeek count value can be generated by leveraging Zeek's value manager object, specifically by calling its `Count` method while providing the value. A time value, however, has to be constructed by `make_intrusive<TimeVal>(time)`.

The `UpdateVector` function is a little more complex as it is designed to accept a vector of any type — in this case integer, unsigned integer, and time — and update a corresponding Zeek vector. Because of this complexity in terms of templates, the function is defined in the header file, as opposed to all other definitions, to simplify compilation. The function template consists of the vector item type, Zeek vector item type, and a function used to create the value itself to address the differences between the instantiation of Zeek values. It works similarly to the primitive value update functions in terms of offset lookup, however, it additionally has to construct a vector type value which has to be populated with Zeek values before assignment. In the excerpt seen in Code Listing 3.10, the method uses the `create_val_func` provided as a parameter to construct individual item Zeek values and assign them to the vector. Only then can be the whole updated vector assigned to the `pstats` record.

Code listing 3.10 UpdateVector function excerpt

```
auto vector_val = make_intrusive<zeek::VectorVal>(vector_type);

for (const T& value : pstats_vector)
    vector_val->Assign(vector_val->Size(), create_val_func(value));

pstats_record->AsRecordVal()->Assign(field_offset,
    std::move(vector_val));
```

A wrapper function `UpdatePstatsVectors` has been created to update each of the four PSTATS vectors by invoking the complex `UpdateVector` function, which would otherwise disturb the organization of `UpdateConnVal` — update of time vector can be seen in Code Listing 3.11.

Code listing 3.11 Zeek time vector update

```
UpdateVector<double, zeek::TYPE_TIME>(pstats_record,
    "pkt_times", pkt_times,[](double value)
    { return make_intrusive<TimeVal>(value); });
```

By implementing both `DeliverPacket` and `UpdateConnVal` overridden methods, the analyzer class is fully capable of handling the analysis previously conducted by the `new_packet` handler in the script-based solution and is expected to significantly enhance performance.

Code listing 3.12 Read packet times vector from Zeek to plugin

```
std::vector<Nemea::UrTime> pkt_times = ReadVector<Nemea::UrTime,
    zeek::TYPE_TIME>(pstats_record, "pkt_times", [](zeek::ValPtr val)
    { return ur_time_from_double(val->AsTime()); });
```

3.2.5 Export

The plugin continues where the script solution could not by introducing the BiF `unirec_export` function, which is made available for use in the script's `connection_state_remove` handler. This wrapper function invokes the plugin's method `UnirecExport`, which conducts three important steps: reads data from the script's `pstats` record, packs the flow data into a UniRec record, and sends it to the output interface. As these steps require a complex approach, all helper functions are declared and defined in separate files, Export.hpp and Export.cc, in an `Export` namespace to support scalability and organization.

3.2.5.1 Read Script Values

Starting with the first task, the loading of Zeek script values from `connection$pstats` record begins by finding the record using its name – same as in `UpdateConnVal` in Section 3.2.4.2. To read individual primitive type values, a template function `ReadValue` is defined to avoid redundant code. This function template consists of a C++ data type, a Zeek data type, and again a function, now to extract the actual value from the Zeek data type. The function retrieves the field from `pstats` record verifies it holds the correct type, and returns its value converted using `convert_func` provided as a parameter. This function is used to retrieve all the custom record fields except for vectors.

For the purpose of reading vector items from any of the four vectors, a template function `ReadVector` is defined. As always, the function starts by retrieving the desired `pstats` field based on its name. It then creates an `std::vector` of the desired type, one by one reads the Zeek vector items, extracts their raw value using `convert_func`, and appends it to the result vector. The biggest challenge lies in constructing a UniRec time type array, as a double type vector cannot be directly assigned to a UniRec time array. The issue is that there is no simple `convert_func` that could take care of `Nemea::UrTime` construction from C++ double, so a helper function `ur_time_to_double` facilitates the conversion. It constructs a `Nemea::UrTime` object, sets its time attribute, and returns it. The `pkt_times` read can be seen in Code Listing 3.12.

As reading all the values is very space-consuming, they are encapsulated in a `ReadData` function, which assigns the read data to a helper `RecordData` struct. At this point, the `UnirecExport` method has all the required data loaded from the main script and is ready to build a UniRec record.

3.2.5.2 Assign to UniRec Record

To start populating a UniRec record, the UniRec output interface kept as a plugin class member variable is leveraged to get a reference to a preallocated record by invoking its `getUnirecRecord` method. To assign count and time values to the record, a template function `AssignValue` is defined. Similarly to the Zeek value assignment, a desired offset in the UniRec record is retrieved using the `ur_get_id_by_name` function, which is then leveraged for the field value assignment using the record's `SetFieldFromType` method.

As IP and MAC addresses require a special approach, they are taken care of by two separate functions: `AssignIpAddr` and `AssignMacAddr`. To assign a UniRec `ipaddr` or `macaddr`, they both have to be constructed using the unirec++ library classes and structs. An IP address

Code listing 3.13 UniRec array reservation an population

```
// Get offset
int id = ur_get_id_by_name(field_name.c_str());

// Populate array
auto dir_arr = record.reserveUnirecArray<T>(src_arr.size(), id);
for (size_t i = 0; i < src_arr.size(); i++)
    dir_arr.at(i) = src_arr.at(i);
```

can be conveniently instantiated from an `std::string` type in which IP addresses are stored throughout the plugin. Unfortunately, MAC addresses can only be created from an `uint8_t` array – essentially unsigned char – which is constructed in `AssignMacAddr` and populated by an `sscanf` function using `%hhx` delimited by a colon format. Both `IpAddress` and `MacAddress` type variables can then be assigned to the UniRec fields also using the record's `setFieldFromType` method.

Vectors are of variable lengths compared to all other defined fields, hence their assignment has to be handled differently. Another dedicated function has been created to address the needs, called `ReserveAndAssignArray`. When the UniRec field offset is retrieved, the record's `reserveUnirecArray` method can be used to reserve memory within the UniRec record. The returned `UnirecArray` object can then be leveraged for a population of the actual record with vector items, as depicted in Code Listing 3.13. To organize the `UnirecExport` function, all assignments are conducted in a new function `AssignData`.

In case any of the implemented assignment functions fails, it simply returns without populating the record while providing an error message — in case `DEBUG` is defined — leaving the field empty. When all assignments have been conducted, the output interface object is used to send the finalized UniRec record by invoking its `send` method. The implemented export satisfies the last extension's requirement, completing the whole hybrid solution.

3.2.6 Compilation

An integral part of the extension development is compilation. It was mentioned before that the plugin-init script prepares the compilation environment, including a configuration script. With new files and libraries introduced to the system, however, CMakeLists.txt has to be reconfigured for the extension to be compiled and linked. As my previous experience with more complex project compilation has been foundational, the process has been a lot of trial and error.

While specifying which new source files — such as Export.cc — should be additionally compiled was easily achievable by adding their relative paths to a `zeek_plugin_cc` function in the CMakeLists.txt file, linking the program with NEMEA's unirec++ library has been a significant challenge. After experimenting with various CMake macros, I have been able to link the library by utilizing `include_directories`, `link_directories`, and `target_link_libraries`. The first two macros were pointed to `/usr/local/include` containing required header files and to `/usr/local/lib` containing required dynamic libraries, respectively. What required the most effort was the last macro, which required a `Nemea_Pstats` target I could not initially identify, a `PUBLIC` keyword necessary to link also against other related targets, and the library that the target needed to link against, unirec++. The reason I could not find the exact target name was that it was not present in the configuration of Make files but was rather generated by CMake commands during compilation. I was able to later identify the target name in CMakeFiles in the plugin's build directory and successfully compile and link the extension. The use of the macros can be seen in Code Listing 3.14.

Code listing 3.14 CMakeLists.txt excerpt - load and link unirec++

```
# Include UniRec library and headers
include_directories(/usr/local/include)
link_directories(/usr/local/lib)

# Link the UniRec library
target_link_libraries(Nemea_Pstats PUBLIC unirec++)
```

Chapter 4
Testing

As the extension is completely implemented, it is necessary to evaluate whether it satisfies all proposed requirements, compare Zeek's performance with and without the extension, and verify whether the produced output is correct. Correctness was tested by comparison of the results with single-flow packet capture and with other Zeek logs. To assess the performance impact of the extension, a set of scripts has been created for single host setup testing.

4.1 Correctness

During the implementation phase, the extension was continuously tested on three sample packet capture files downloaded from Tcpreplay, ranging from 37 to 40 686 flows [15]. To test the final version for correctness, I started by focusing on single-flow files to isolate eventual errors better and for overall clarity. Both extraction and subsequent comparison were conducted using Wireshark protocol analyzer [16].

I began by focusing on the Zeek script base shared by both solutions, which starts by defining a custom record and assigning it to `connection`. I extracted a simple nine-packet HTTP flow from the testing packet capture, and in `new_connection` event handler, I added two print statements printing the connection object before and after `pstats` assignment in its entirety, including all its initialized and uninitialized fields. When observing Zeek's output after reading the one-flow file, I observed an uninitialized `pstats` field at the end of the connection in the first output, which means the record was correctly redefined. When the connection was printed after the assignment, the output already contained the initial values provided during `pstats` instantiation, which I compared to the packet capture. As both IP addresses and port numbers matched the values seen in Wireshark, the record instantiation also works as expected.

4.1.1 Script

Before moving to the final hybrid solution, I tested the script version as its correctness is necessary for an accurate performance comparison of the two solutions. The solutions start to differ when it comes to the individual packet inspection, so I started by examining the behavior of the script's `new_packet` event handler, specifically packet timestamps, directions, and bytes – TCP flags are extracted only after script parameters are checked, which will be tested later. The HTTP flow consists of nine TCP packets, beginning with a 3-packet TCP handshake, followed by a GET request and an acknowledgment packet, a 304 code response again followed by acknowledgment, and a 2-packet TCP connection termination, so it is expected that the handler will be called for each. To verify it is called as expected, I added a print statement to the very top of the handler.

As I observed nine lines of output, it is ensured that the event handler is indeed executed with every packet.

To check whether the handler correctly extracts and assigns all desired fields, I decided to print the whole final `pstats` record at the end of the connection termination handler before logging to avoid browsing extensive output. A first important observation is that each of the PSTATS vectors only contained two values, which confirms the `LogEmpty` parameter set to false works as expected, as only two packets in the flow carry a transport-layer payload. Setting it to true resulted in all nine packets being logged in each vector. To confirm the second parameter `LogFirst` also functions properly, I left the `LogEmpty` parameter set to true while configuring the `LogFirst` to number 7, which correctly resulted in 7 items in each vector upon flow's termination.

As Zeek timestamps are represented in seconds with microsecond precision — seconds since 1970-01-01 — the Wireshark default time offset view was changed to match the format for comparison. With the unified format, I was able to confirm the individual packet times were both correctly extracted and assigned. The rest of the `pstats` fields' values were also successfully compared to the packet capture — connection time boundaries were also compared to conn.log for consistency across Zeek's environment – ensuring the extension conducts a comprehensive and correct analysis. Similar testing was also conducted for UDP, and ICMP flows to address all possible protocol cases.

The last remaining function to be tested is the ASCII logging. For all the mentioned protocols, I simply compared the verified printed `pstats` record before logging and the contents of pstats.log. As all fields matched the printed values, the correctness testing of the script solution is complete.

4.1.2 Plugin

To test all functionality conducted by the compiled plugin, it is first necessary to ensure that Zeek correctly recognizes it. By executing `zeek -NN`, Zeek prints all available plugins in a verbose mode. The output in my case is positive, as the plugin is not only listed but includes a custom-defined analyzer, two BiF functions `unirec_init` and `unirec_export`, and a statement that `SetupAnalyzerTree` hook is implemented.

With the plugin installed and the components registered, I began testing the plugin itself. I began with the plugin configuration, specifically with the reading of script parameters by adding temporary print statements after each parameter read. By specifying each of the three plugin parameters, I observed whether changes were visible in the plugin environment. With control over the `OutIfcSpec` parameter, I continued with the `unirec_init` function additionally called in the main.zeek script, which sets up the UniRec interface.

I utilized a logger NEMEA module, which I configured to subscribe to localhost at TCP port 1111 in its most verbose mode (`-vvv`), which enabled me to observe unsuccessful connections every second. As I executed Zeek while specifying TCP port 1111 as an output interface, I immediately noticed a successful connection from the logger module, which means the UniRec interface is being correctly initialized and we can move to the analysis testing.

To test the data collection, I chose the same approach of printing the `c$pstats` record in its entirety after the data collection is finalized rather than with every packet — even in the case of a 9-packet flow, the output can be very extensive and illegible. By comparing the output with the verified script solution output, I verified that the analysis was conducted correctly and all collected information was accurate.

The last step is to ensure that data is correctly packed into a UniRec record and received without any malformation. As the logger module simply logs all received records and the UniRec template closely resembles the record defined in the Zeek script, as soon as the received UniRec message gets printed, it can be conveniently compared. By comparing each field in the pstats.log to the logger's output, I ensured all values and vectors were transmitted without corruption and verified the correctness of the last extension's functionality.

Code listing 4.1 lscpu command output excerpt

```
L1d:                    512 KiB (12 instances)
L1i:                    512 KiB (12 instances)
L2:                     12 MiB (9 instances)
L3:                     25 MiB (1 instance)
```

4.2 Performance

As organizations typically do not dedicate special machines for flow extraction and collection, the performance testing was conducted on a single host — a 64-bit Debian desktop machine, 12th Gen Intel(R) Core(TM) i7-12700K CPU with 20 cores and the following cache metrics 4.1, and 32GB of memory.

The approach was to conduct multiple measurements by generating traffic consisting of gradually decreasing size packets and capturing it using a Zeek instance first in its default mode, then with the script solution active — only analysis and logging without UniRec export — and finally with the hybrid extension. Over a period of time, I wanted to observe the received on a network interface, the number of packets Zeek processed, and most importantly, the difference between them to determine the advantage of the plugin.

Starting with the traffic generation, I explored several network testing tools including tcpreplay, however, for its multithreading capabilities and granular traffic generation options, I selected iperf3 [17]. This command line tool works in 2 modes: client mode for generation of network traffic and server mode for accepting it. The most important options are bandwidth, which was set to 3 Gbps, and the length of individual packets in bytes, which I leveraged for the gradual packet increase while keeping bandwidth static.

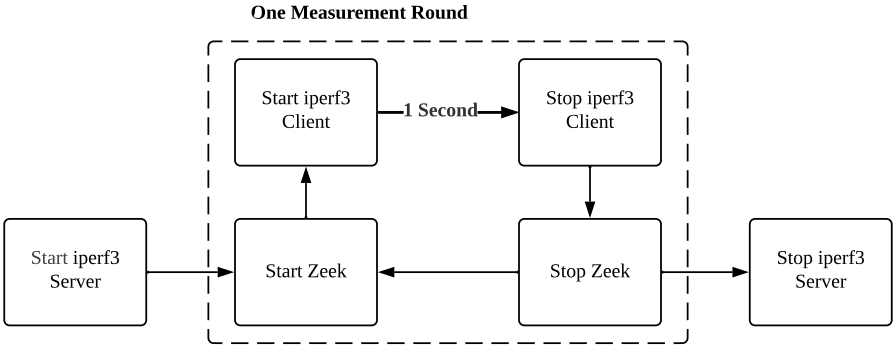

Figure 4.1 Performance testing design

To automate the measurement process, I created a measure.sh Bash script, designed in Figure 4.1, which begins by starting an iperf3 server on CPU core 0 using taskset command, as the server is only single-threaded. Specifying cores for each of the measurement tasks — iperf3 server, iperf3 client, and Zeek instance — ensures that the performance of one is not affected by the other and results are as accurate as possible. Next, measurement rounds are conducted while the provided packet length — continually decreased by the provided step size — is greater than 16 bytes, the minimum acceptable value for the iperf3 packet length parameter. To initially avoid fragmentation when testing the performance over the network before turning to the loopback interface approach, the starting packet length was set to 1472 bytes. Although it could be set to up to 64KB, the performance is expected to degrade, especially with lower values, as much more packets per second have to be generated to reach the set bandwidth, so by keeping the initial

value of 1472, redundant measurement rounds are avoided. Selecting a 28-byte reduction step size ensures that the 53rd measurement is conducted with a 16-byte packet length, reaching the biggest performance load possible.

In each round, I started by reading the number of packets currently received on a loopback interface by referring to system file /sys/class/net/lo/statistics/rx_packets, which is later used to calculate how many packets were actually received. Then, I continued by launching a Zeek instance first in its default configuration on core 1 while executing the iperf3 client on 18 threads 2-19. The iperf3 was executed using the timeout command, letting it run for exactly one second each round. Additionally, it was started in a UDP mode to avoid confirmation packets on the interface, providing localhost as a destination, 3Gbps bandwidth, and calculated packet length, which is at the same time logged to packet_sizes.log for later data visualization. After sending traffic for one second, the zeek instance was first terminated before accessing the interface `rx_packets` again to calculate the difference, which is appended to interface_packets.log.

When exploring options to get the number of packets processed by Zeek, I initially noticed that when a Zeek instance is terminated, Zeek informs how many packets were dropped and not processed. Although these numbers might seem very handy for the purpose of packet loss measurement, after some testing, I determined they are very inaccurate. For this reason, I leveraged a total number of received packets by Zeek also provided upon its termination, appending it to zeek_packet.log.

After conducting the 53-round measurement for the default configuration, I conducted the whole measurement again for both the incomplete script solution and the hybrid solution. With Zeek and interface packet and Zeek packet pairs of logs for each configuration, I moved to Jupyter notebook [18] for collected data visualization in Python language.

Leveraging pandas library [19], I started by reading the common packet_sizes.log for all measurements and all configuration-related packet count logs. As all logs only contain one column, I utilized a `read_csv` pandas function to read them into data frames. I concatenated the two logs and the packet sizes for each solution into `default_data`, `script_data`, and `plugin_data`. Using matplotlib library [20], I visualized each dataset by creating scatter plots that plot packet sizes against processed packet numbers 4.2.

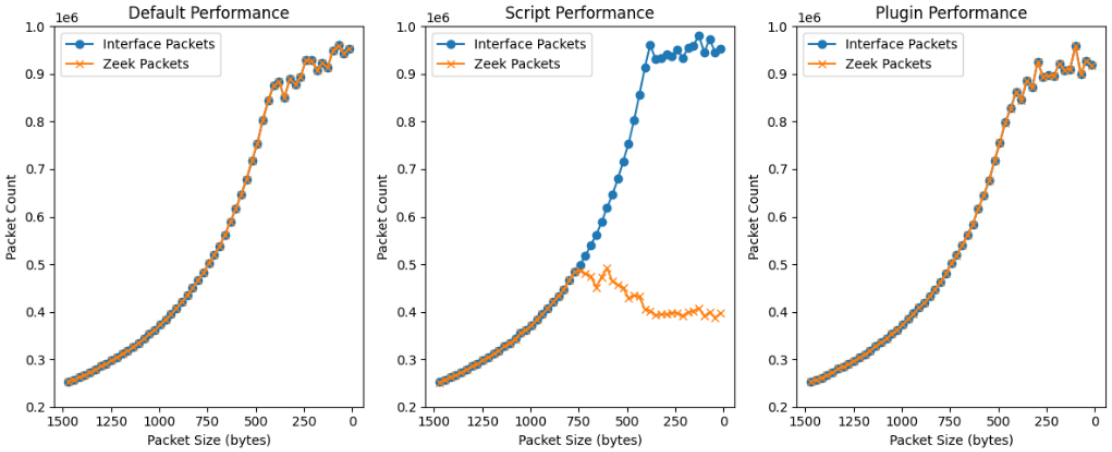

Figure 4.2 Comparison of solutions' performance to Zeek default configuration

We can see from the graph that the number of packets handled by iperf3 stagnates somewhere below one million packets per second, which might be a result of the performance limitations of either iperf3 or the testing machine itself. An important observation is that the script solution results in a significant loss when packets per second reach approximately 400 thousand. However, the plugin solution exceeded expectations and did not affect the overall Zeek performance at all.

Conclusion

This thesis has described the requirements, design, and development of the Zeek extension aimed at extended flow statistics export and integration with the NEMEA system. Before delving into technicalities, related themes were described so readers can better grasp the challenges and decisions encountered throughout the work.

By exploring the field of network monitoring with a focus on flow-based monitoring architecture, Zeek's and NEMEA's capabilities and structure, and the workings of IPFIXprobe flow exporter, I was able to lay down the foundational requirements for the extension. Through research of the existing solutions and possibilities for Zeek extension development, I have identified both the strong sides and shortcomings of Zeek's custom scripting language and compiled plugin solutions. Throughout the design of the sole script solution, however, I determined that UniRec export requirements cannot be satisfied. Another challenge faced when designing individual packet inspection in the script was that high-level script events generated with every packet in a high throughput network can be very resource intensive, which led to the conclusion that the extension has to employ compiled components.

Since the script solution overlaps with the plugin, specifically with its main Zeek script, the implementation focused on both solutions for the possibility of subsequent performance comparison. The development of the script solution required an understanding of Zeek's scripting language, which was not too complicated, thanks to comprehensive documentation. The compiled approach, on the other hand, posed a significant challenge. From understanding Zeek's C++ API to linking third-party libraries, it has been a process of continuous learning through a lot of trial and error. Since open-source projects — both Zeek and NEMEA — are often work in progress, the development required browsing through Github source code repositories, and in my hardest times, led me to reach out to the open-source community, which has been invaluable.

After completing the designed solutions, I started by testing both for correctness through comparison of the results with packet capture files and other logs produced by Zeek for consistency across the environment. I then turned to a comparison of Zeek's performance in its default configuration, with the script active and with the plugin installed. It was confirmed that the Zeek script can pose a large overhead and thus cause information loss. The plugin solution exceeded all expectations as it did not have an impact on Zeek's performance.

In the future, the extension could be further enhanced by implementing dynamic, runtime-configurable template UniRec records, a feature supported by NEMEA but not yet leveraged, as the UniRec record format is defined statically. This capability would increase the extension's adaptability, providing custom solutions to diverse network environments.

Bibliography

1. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*. 2013. Available also from: https://www.ietf.org/rfc/rfc7011.txt. [online] referenced on 13.3. 2024.
2. HOFSTEDE, Rick; CELEDA, Pavel; TRAMMELL, Brian; DRAGO, Idilio; SADRE, Ramin; SPEROTTO, Anna; PRAS, Aiko. Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *IEEE Communications Surveys & Tutorials*. 2014, vol. 16, no. 4, pp. 2037–2064. ISSN 1553-877X. Available from DOI: 10.1109/comst.2014.2321898.
3. *Zeek*. 2024. Available also from: https://docs.zeek.org/en/master/index.html. [online] referenced on 5.3. 2024.
4. *NEMEA Github*. 2016. Available also from: https://github.com/CESNET/NEMEA. [online] referenced on 7.3. 2024.
5. *Network Traffic Analysis*. 2024. Available also from: https://www.rapid7.com/fundamentals/network-traffic-analysis/. [online] referenced on 13.3. 2024.
6. *Cisco Netflow V5*. 2024. Available also from: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/fnetflow/configuration/15-mt/fnf-15-mt-book/fnf-v5-export.pdf. [online] referenced on 2.5. 2024.
7. *Cisco Netflow V9 RFC*. 2004. Available also from: https://www.ietf.org/rfc/rfc3954.txt. [online] referenced on 2.5. 2024.
8. *Suricata Analysis and Detection SW*. 2024. Available also from: https://docs.suricata.io/en/latest/index.html. [online] referenced on 2.5. 2024.
9. CEJKA, Tomas; BARTOS, Vaclav; SVEPES, Marek; ROSA, Zdenek; KUBATOVA, Hana. NEMEA: A framework for network traffic analysis. In: *2016 12th International Conference on Network and Service Management (CNSM)*. IEEE, 2016. Available from DOI: 10.1109/cnsm.2016.7818417.
10. ZIMMERMANN, H. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*. 1980, vol. 28, no. 4, pp. 425–432. ISSN 0096-2244. Available from DOI: 10.1109/tcom.1980.1094702.
11. *Ipfixprobe GitHub*. 2023. Available also from: https://github.com/CESNET/ipfixprobe/tree/master. [online] referenced on 2.5. 2024.
12. *Zeek Github Repository*. 2024. Available also from: https://github.com/zeek/zeek/tree/master/. [online] referenced on 11.4. 2024.
13. *TRAP Interface Specifier*. 2022. Available also from: https://nemea.liberouter.org/trap-ifcspec/. [online] referenced on 23.4. 2024.

14. *NEMEA Framework Github*. 2024. Available also from: https://github.com/CESNET/Nemea-Framework. [online] referenced on 23.4. 2024.
15. *Tcpreplay Sample Captures*. 2017. Available also from: https://tcpreplay.appneta.com/wiki/captures.html. [online] referenced on 27.4. 2024.
16. *Wireshark*. 2024. Available also from: https://www.wireshark.org/. [online] referenced on 27.4. 2024.
17. *Iperf3 for Network Testing*. 2023. Available also from: https://iperf.fr/. [online] referenced on 30.4. 2024.
18. *Jupyter Notebook*. 2024. Available also from: https://jupyter.org/. [online] referenced on 5.5. 2024.
19. *Pandas Python Library*. 2024. Available also from: https://pandas.pydata.org/. [online] referenced on 5.5. 2024.
20. *Matplotlib Python Library*. 2023. Available also from: https://matplotlib.org/. [online] referenced on 5.5. 2024.

Contents of the attachment

- `README.md` — Brief media content description
- `pstats.zeek` — Script solution
- `nemea/` — Plugin solution
  - `scripts/` — Plugin-associated scripts
  - `src/` — C++ and BiF source files
  - `configure` — Compilation configuration script
  - `CMakeLists.txt` — Source file and library list for CMake
  - `Makefile` — Makefile
  - `README` — Mandatory plugin README
  - `VERSION` — Plugin version
  - `COPYING.txt` — BSD-style licence
- `pcaps/` — Testing packet captures
- `performance/` — Performance measurement scripts and visualization
- `text/`
  - `thesis.pdf` — Thesis in PDF format
  - `thesis.zip` — Thesis source code in LaTeX format