UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LUCAS SONNTAG HAGEN

An Automatic Code Generation Mechanism for Enabling Intrusion Detection Systems at Data-Plane Speeds: A Zeek-based Study

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Advisor: Prof. Dr. Luciano Paschoal Gaspary
Co-advisor: Dr. Jonatas Adilson Marques

Porto Alegre
August 2022

ACKNOWLEDGEMENTS

I would like begin thanking my parents Rejane Sonntag Hagen and Roberto Hagen for providing me with a quality education and supporting me at all times. Also to my brother Bruno and my girlfriend Giulia who were part of this trajectory, encouraging me to move forward and not to give up.

To my advisor, Prof. Dr. Luciano Paschoal Gaspary, who guided me in this difficult task called TCC, always with patience and much wisdom, sharing a bit of all his knowledge and experience. To my co-advisor Dr. Jonatas Adilson Marques and to MSc. Alexandre da Silveira Ilha for sharing their time answering my questions and guiding me in my research.

I also thank the institution UFRGS and its professors, where I started a professional journey, met great friends, and built my resume. I am immensely grateful for all the opportunities I have received. I would like to briefly mention professors Raul and Taisy Weber, Lucas Schnorr, Gabriel Nazar, and Mara Abel, who had a special participation in this academic journey.

Furthermore, I would like to thank my friends and colleagues, who have been very important in my academic journey. Unfortunately there are too many names to mention and I would certainly forget someone.

As for the exchange program in Kaiserslautern, I thank my supervisor Prof. Dr. Reinhard Gotzhein and the doctoral student MSc. Paulo Aragão, who welcomed and mentored me during my time at the Technische Universität Kaiserslautern. I especially mention my friends Bernardo, Emily, and Iron, who became a family in the year 2020, during the height of the COVID-19 Pandemic.

Furthermore, I would like to mention my internship supervisors and other colleagues, with whom I shared learning and leisure moments: Lauro Souza and Breno Araújo, from my internship at Google Brazil in the year 2021-2022; Daniel Thiel, Hélio Fuques, and Hernandi Krames from my internship at AEL Sistemas in the year 2018-2020.

Finally, I thank the other professors, mentors, family members, and friends, who directly and indirectly participated in my academic journey, transmitting learning, sharing ideas and good moments, sharing moments of anguish and also of joy.

AGRADECIMENTOS

Agradeço aos meus pais Rejane Sonntag Hagen e Roberto Hagen por me proporcionarem uma educação de qualidade e me apoiarem em todos os momentos. Também ao meu irmão Bruno e minha namorada Giulia que fizeram parte dessa trajetória, incentivando-me a seguir em frente e a não desistir.

Ao meu orientador, Prof. Dr. Luciano Paschoal Gaspary, que me guiou nessa difícil tarefa chamada TCC, sempre com paciência e muita sabedoria, compartilhando um pouco de todo o seu conhecimento e sua experiência. Ao meu co-orientador Dr. Jonatas Adilson Marques e ao MSc. Alexandre da Silveira Ilha por compartilharem seu tempo sanando as minhas dúvidas e guiarem-me em minha pesquisa.

Agradeço também à instituição UFRGS e seus professores, onde iniciei uma jornada profissional, conheci grandes amigos e construí meu currículo. Sou imensamente grato por todas as oportunidades recebidas. Faço aqui breve menção dos professores Raul e Taisy Weber, Lucas Schnorr, Gabriel Nazar e Mara Abel os quais tiverem uma participação especial nessa jornada acadêmica.

Além disso, gostaria de agradecer aos meus amigos e colegas de curso, os quais foram muito importantes em minha trajetória acadêmica. Infelizmente são muitos nomes e certamente esqueceria de alguém se tentasse mencioná-los.

Quanto ao intercâmbio realizado em Kaiserslautern, agradeço ao meu orientador Prof. Dr. Reinhard Gotzhein e ao doutorando MSc. Paulo Aragão, que me acolheram e mentoraram a minha passagem pela Technische Universität Kaiserslautern. Menciono, em especial, meus colegas de intercâmbio Bernardo, Emily e Iron, os quais se tornaram uma família no ano de 2020, durante o auge da Pandemia da COVID-19.

Ademais, gostaria de mencionar os meus supervisores de estágios e outros colegas de trabalho, com os quais dividi momentos de aprendizado e de lazer: Lauro Souza e Breno Araújo, do estágio que realizei no Google Brasil no ano de 2021-2022; Daniel Thiel, Hélio Fuques e Hernandi Krames do estágio na AEL Sistemas no ano de 2018-2020.

Por fim, agradeço aos demais professores, mentores, familiares e amigos, que direta e indiretamente participaram da minha jornada acadêmica, transmitindo aprendizados, compartilhando ideias e bons momentos, dividindo momentos de angústia e também de alegria.

ABSTRACT

Zeek is an Intrusion Detection System (IDS) based on events and policy interpreters. Currently, these events are triggered in a general-purpose CPU, which analyses all incoming packets. This process has become increasingly prohibitive with the rapidly increasing network speeds. In this work, we propose an automatic code generation mechanism to offload IDS operations to Programmable Data Planes (PDP). The mechanism is capable of identifying the requirements for the desired set of Zeek Scripts, and, using reusable templates, automatically generates code for the P4 switch and a Zeek Plugin. Templates specify how a Zeek Event is offloaded to PDPs and can be reused for different scripts that rely on the same events. The generated code offloads the initial packet filtering and event identification procedure using hardware acceleration. The existing IDS engine then analyzes and processes the triggered events. The proposed solution results in low development costs demanded of a human operator while substantially alleviating resource usage by Zeek.
Keywords: Zeek. P4. Programmable Data Planes. Security. Code Generation.

Um Mecanismo de Geração Automática de Código para Permitir Sistemas de Detecção de Intrusão em Velocidades de Planos de Dados: Um Estudo Baseado no Zeek

RESUMO

Zeek é um Sistema de Detecção de Intrusão (IDS) baseado em eventos e interpretadores de políticas. Atualmente, esses eventos são processados em um processador de propósito geral, que analisa todos os pacotes recebidos. Esse processo tornou-se cada vez mais proibitivo com o rápido aumento das velocidades das redes. Neste trabalho, propomos um mecanismo de geração automática de código para descarregar operações IDS para Planos de Dados Programáveis (PDP). O mecanismo é capaz de, usando templates reutilizáveis, identificar os requisitos para o conjunto desejado de scripts Zeek e gerar automaticamente código para o switch P4 e um Plugin Zeek. Os templates especificam como um evento Zeek é descarregado para PDPs e podem ser reutilizados para diferentes scripts que dependem dos mesmos eventos. O código gerado descarrega a filtragem inicial de pacotes e a identificação de eventos usando aceleração de hardware. O mecanismo IDS existente analisa e processa os eventos acionados. A solução proposta resulta em baixos custos de desenvolvimento exigidos de um operador humano enquanto alivia substancialmente o uso de recursos pelo Zeek.

Palavras-chave: Zeek. P4. Planos de Dados Programáveis. Segurança. Geração de Código.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
BMv2	Behavioral Model Version 2
C2	Command and Control
DDoS	Distributed Denial of Service
DDR	Double data rate
DoS	Denial of Service
DPD	Dynamic Protocol Detection
DSL	Domain-specific Language
EE	Zeek Event Engine
FPGA	Field Programmable Gate Array
FTP	File Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
MT/s	Megatransfers per Second
NIDS	Network Intrusion Detection System
NPU	Network Processing Unit
NTP	Network Time Protocol
NVMe	Nonvolatile Memory Express

PAF	Packet Analysis Framework
PCAP	Packet Capture
PDP	Programmable Data Plane
PDU	Protocol Data Unit
PFD	Programmable Forwarding Device
POF	Protocol Oblivious Forwarding
PPS	Packets Per Second
PSI	Zeek Policy Script Interpreter
RAM	Random Access Memory
RNA	Reconfigurable Network Analytics
RQ	Research Question
SDN	Software Defined Network
SDN	Software Defined Network
SSD	Solid State Drive
TCP	Transmission Control Protocol
TTL	Time-To-Live
UDP	User Datagram Protocol
UID	Unique Identifier
VoIP	Voice over Internet Protocol

CONTENTS

1 INTRODUCTION

In this globally connected world, more people depend daily on the Internet for many tasks of their lives. This creates two challenges: rapidly increasing network speeds and traffic, and more attack targets for malicious actors, resulting in a rise in computer security incidents and an increase in the difficulty of detecting these attacks. According to Akamai Technologies (2022), ransomware attacks caused 20 billion US dollars of damage globally in 2021. Netscout (HUMMEL; HILDEBRAND, 2021) estimates the potential revenue loss from Distributed Denial of Service (DDoS) extortion of Voice-over-IP (VoIP) providers was between 9 and 12 million US dollars in that same year.

Various systems are used to mitigate and detect attacks, including Intrusion Detection Systems (IDS) and Network Intrusion Detection Systems (NIDS), e.g., Zeek (PAXSON, 1999), Suricata (The Open Information Security Foundation, 2022), and Snort (Cisco Systems, 2022). These systems have one common problem: the difficulty of detecting attacks and threats with high accuracy and high performance, especially given the large volume of data modern network infrastructures are capable of transmitting (at high rates). The root of the problem lies in the type of hardware architecture these systems use, which are general-purpose servers that require copying data from the network to memory and then manipulating it. This is incompatible with the speeds we observe today. In this scenario, together with the emergence of Software Defined Networks (SDNs) and Programmable Data Planes (PDPs), there is an opportunity to execute intrusion detection tasks directly in Programmable Forwarding Devices (PFDs) at line rate. For instance, Ilha (2022) uses the benefits of PDPs to offload specific IDS tasks within the scope of some case studies. This approach, although functional, needs to be manually extended for each additional monitoring scenario, which requires the work of skilled software developers.

In this work, seeking to broaden the range of IDS tasks we can delegate to a Programmable Data Plane, we propose an approach that facilitates the offloading of specific tasks, typically executed in general-purpose processors, to a PFD. More importantly, our design enables fully-automated integration between IDS and PDP, thus eliminating the need for a network operator to develop an *ad-hoc* approach for each specific offloading task. To accomplish this, we propose additions to the RNA Framework by introducing a mechanism that can parse Zeek Scripts, identify the operations to be offloaded, and, based on templates, generate a complete approach to offload a set of Zeek scripts to PDPs.

The remainder of this manuscript is organized as follows: in Chapter 2, we briefly describe the "anatomy" of the attacks studied in this project, the Zeek Network Security Monitor System and Programmable Data Planes, focusing on the P4 programming language. In Chapter 3, we present the architecture our project is based on, the Reconfigurable Network Analytics framework (ILHA, 2022), and describe some of our additions to that framework. In Chapter 4, we propose an automatic code generation mechanism for offloading Zeek Scripts to PDPs. In Chapter 5, we describe some implementation aspects of our prototype and evaluate the capabilities and performance of our approach. To finalize, in Chapter 6, we conclude the text by summarizing the results and presenting suggestions for future work.

2 BACKGROUND

In this chapter, we briefly describe the main fundamental concepts and technologies related to this work. Specifically, we describe the monitoring scripts we considered during the design and evaluation, the Zeek network security monitoring system, and the P4 language for Programmable Data Planes (PDP).

2.1 Monitoring Scripts

In this section, we describe the monitoring scripts used to later evaluate our proposals of Chapters 3 and 4. They were chosen based on simplification assumptions we made to limit the scope of the project. The main assumption was that monitoring scripts should not require stateful tracking, e.g., it should not be required to track if TCP connections have been successfully established.

FTP Bruteforce Attack

FTP Bruteforce attacks are a common method of gaining unauthorized access to FTP servers. This attack consists of multiple and coordinated attempts to log on to a server, each time trying a new potential password. An attacker tries all or various combinations of passwords until one of them is accepted, effectively discovering someone's password (ADAMS, 2019).

NTP Monlist

An NTP Monlist attack is an amplification reflection-based volumetric DDoS attack, which exploits vulnerable Network Time Protocol (NTP) servers to attack other services. Using IP-spoofed NTP queries, an attacker can lead exploited servers to generate a large amount of traffic to a victim and overload its network (Cloudflare, Inc., 2022).

ICMP Pingback Tunnel

Pingback is a name given to a malware that uses ICMP messages to tunnel communications between infected hosts and command and control (C2) servers. This gives attackers a covert communication channel between C2 servers and infected hosts. Ana-

lyzing network traffic enables network operators to identify compromised hosts and block the operation of the malware (TAVARES, 2021).

Traceroute

Traceroute, differently from the above-mentioned attacks, is not an attack but a network diagnosing tool. It can determine, using multiple *low-TTL* packets, the route a packet took to reach a destination (GUPTA, 2021). Even not being an attack, we mention traceroute here because it will be one of the monitored behaviors in our evaluation in Chapter 5.

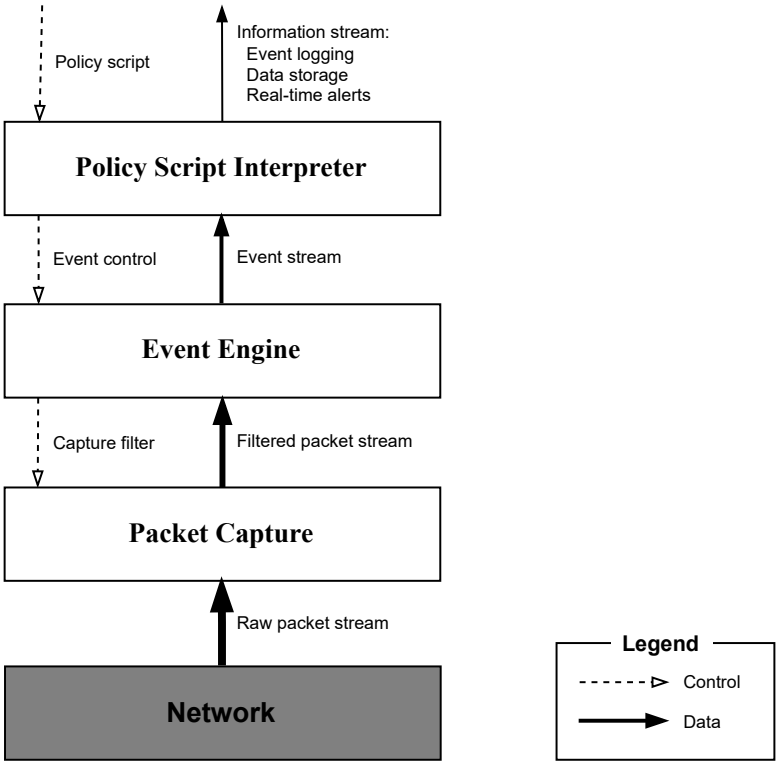
2.2 The Zeek Network Security Monitoring System

A Network Intrusion Detection System (NIDS) is a system that monitors a computer network and is capable of generating alerts for network operators (Science Direct, 2022). These alerts help to prevent future and ongoing cyberattacks. Currently, there are a number of different NIDS solutions, which include Snort (Cisco Systems, 2022), Suricata (The Open Information Security Foundation, 2022), and Zeek (The Zeek Project, 2022b). We focus on Zeek because of its open-source nature, and architecture focused on extensibility and performance. Furthermore, it was the selected NIDS by Ilha (2022) for the original RNA, which is the basis of our approach described in Chapters 3 and 4.

The Zeek Network Security Monitoring System (The Zeek Project, 2022c) is an Intrusion Detection System originally named *Bro*, which was proposed by Paxson (1999). Zeek is an IDS focused on high-speed monitoring, real-time notification, extensibility, and clear separation between mechanism and policy (PAXSON, 1999). It was developed in a layered architecture shown in Figure 2.1, which also enables the distribution of its processing to account for high throughput networks. The arrows represent incoming data, and the dotted lines represent control and management communication.

The first component of this architecture, from a bottom-up perspective, is the *Packet Capture*. The *Packet Capture* layer filters incoming packets from the network, ensuring only packets of interest reach the next layer, which is the *Event Engine* (EE). The *Event Engine* introduces semantic value to packets, translating them into events. The final layer, the *Policy Script Interpreter* (PSI), receives the event stream produced by the EE and interprets these events, generating logs and real-time alerts, or *notices*, as Zeek calls them. We now describe in more detail each component.

Figure 2.1 – Zeek Architecture



Source: Ilha (2022), adapted from Paxson (1999)

Packet Capture

The *Packet Capture* layer provides an abstraction layer between the network and the *Event Engine*. The original Zeek proposal (PAXSON, 1999) was composed only by the *libpcap* library (JACOBSON; LERES; MCCANNE, 1994). With this extra layer, Zeek is isolated from link-layer technologies, has improved compatibility with other Unix systems, and becomes able to read from packet traces saved as *PCAP* files.

Event Engine

The *Event Engine* is the most important layer for our framework (presented in Chapter 3) and where our solution is executed. It is responsible for parsing incoming packets and generating events corresponding to those packets. As an example, we cite parsing an incoming ICMP Echo Request and triggering the `icmp_echo_request` event with all the required structured information associated with it.

The internal structure of the *Event Engine*, as explained by Ilha (2022), can be divided into four stages: acquisition, packet analysis, session analysis, and application

layer parsing. In the acquisition stage, packets sent by the *Packet Capture* component are received and forwarded to the next stage. In the packet analysis stage, the Packet Analysis Framework (PAF) receives the packets as Protocol Data Units (PDUs) and processes each layer of the packet. In this stage, where lower-layer protocols are analyzed, there is still a clear definition of headers and payloads, and the headers usually contain information of what is the next layer's protocol. Using this information, the PAF processes each header, extracting information and delivering its payload to the next layer until the transport layer is reached. Each analysis layer in the Packet Analysis Framework is called an *Analyzer*. Developers can also create new Analyzers and use them to process new protocols and create different events.

Still in the Event Engine, the session analysis stage creates a *Connection* object that, despite the name, does not only represent a connection as in network terminology but also flows and sessions, for example, ICMP Echo Requests and Replies. This session is then stored and used to associate requests and replies, facilitating parsing for the next stage. After the *session* has been created, the application layer parser selects possible application layer analyzers based on well-known port numbers. Since services are not strictly bound to specific ports, the application layer parser is able to dynamically identify application layer protocols using Dynamic Protocol Detection (DPD), which was introduced by Dreger et al. (2006). Throughout the whole execution of the *Event Engine*, it generates events that have semantic value, and that will be handled by the Policy Script Interpreter (PSI). These events vary from lower-level events, such as a TCP connection established, with the `connection_established` event, up to application-layer protocols, for example, an FTP Request and Reply, represented respectively by the `ftp_request` and `ftp_reply` events.

Policy Script Interpreter

The *Policy Script Interpreter* is an event-driven script interpreter. It processes the events that were triggered by the *Event Engine* using scripts written in a domain-specific language called ZeekScript. To process events, scripts define *event handlers*, similar to functions that process the events to generate logs and alerts. Zeek comes with multiple built-in scripts, such as those for FTP Bruteforce detection, traceroute detection, and others. Users can also write scripts and run their policies.

Candidate Operations for Offloading

The Zeek Network Monitoring System, as described above, has three different components that could be considered for offloading. Some of those components are more suitable than others to be offloaded to Programmable Data Planes. The Policy Script Interpreter requires interpretation and text parsing capabilities, which, while not something very complex, are demanding and are not able to execute in the specific processing capability of a PDP device. Unlike this component, the Packet Capture and the Event Engine perform tasks well suited for the type of processing executed in Programmable Data Plane devices. As Ilha (2022) describes in his thesis, the best candidate operations for PDP offloading are those related to the Packet Capture and Event Engine layers. In these layers, the best suitable operations are parsing protocols up to the transport layer (Ethernet, IPv4, UDP, and others), packet filtering, and lightweight packet inspection.

2.3 Programmable Data Planes and the P4 Language

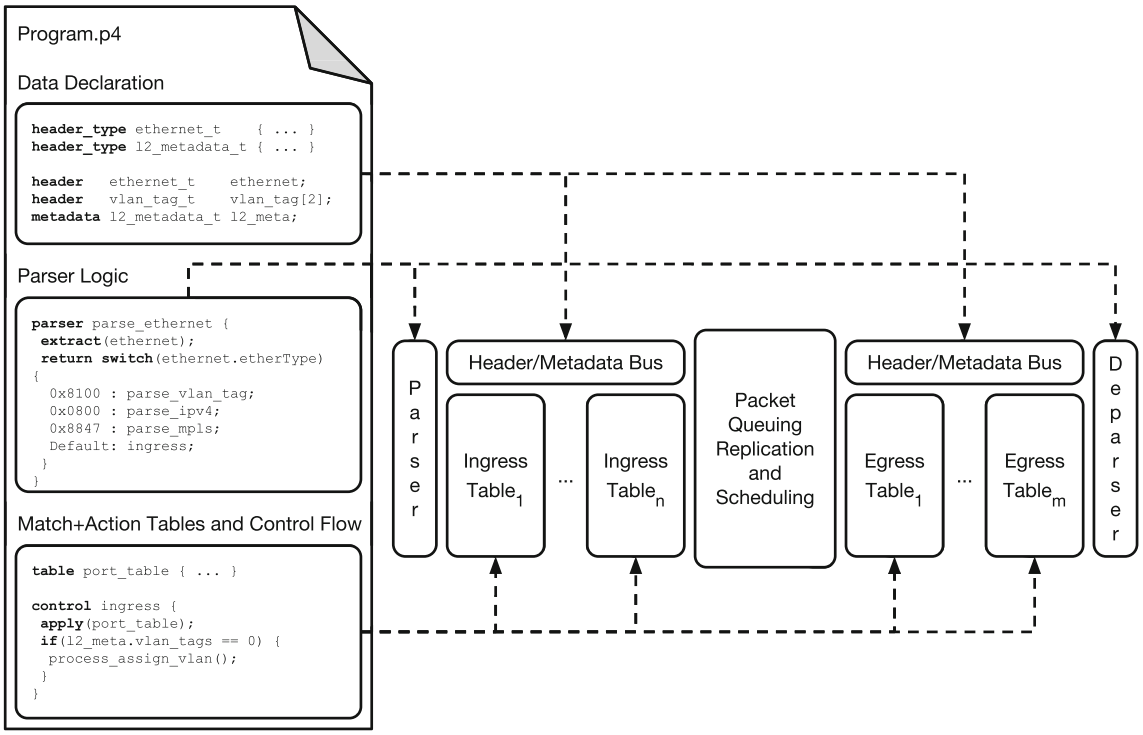
Programmable Data Planes emerged as a solution to program Software Defined Network (SDN) devices even further, allowing network operators to define protocols and network flow, effectively programming them. Domain-specific languages (DSLs) were developed to program these devices, two notable mentions are POF (SONG, 2013) and P4 (BOSSHART et al., 2014). In this project, we focus on P4.

P4 stands for *Programming Protocol-Independent Packet Processing*. It was proposed by Bosshart et al. (2014) and is a domain-specific language for programming network devices. P4 provides an abstraction for packet parsing and processing by providing a generalized forwarding model (CORDEIRO; MARQUES; GASPARY, 2017). It is also target-independent and can be executed in different types of switches.

A P4 program is organized into three sections: (a) *data declaration*, (b) *parser logic*, and (c) *match+action tables and control flow* (CORDEIRO; MARQUES; GASPARY, 2017). These sections can be seen in Figure 2.2. The *data declaration* section defines all required data structures: the headers and the meta-data. These structures are mapped to a header and meta-data bus, which is used throughout the whole pipeline. The definition of a P4 structure is similar to a *struct* definition in C, i.e., a structure can contain multiple data fields, and each field has a type and a length. Differently from C, whose types have sizes expressed in bytes, P4 type sizes are defined by the number of bits. The

parser logic specifies how packets are parsed and deparsed. This definition uses a state machine to define parsing states, progressing from one protocol to another. Finally, the *match+action tables and control flow* defines the control flow of the ingress and egress pipelines. In this section, it is possible to define routing rules and forward packets to specific ports. This is done using *match+action* tables, which match specific header field values to actions to be executed.

Figure 2.2 – P4 Forwarding Model

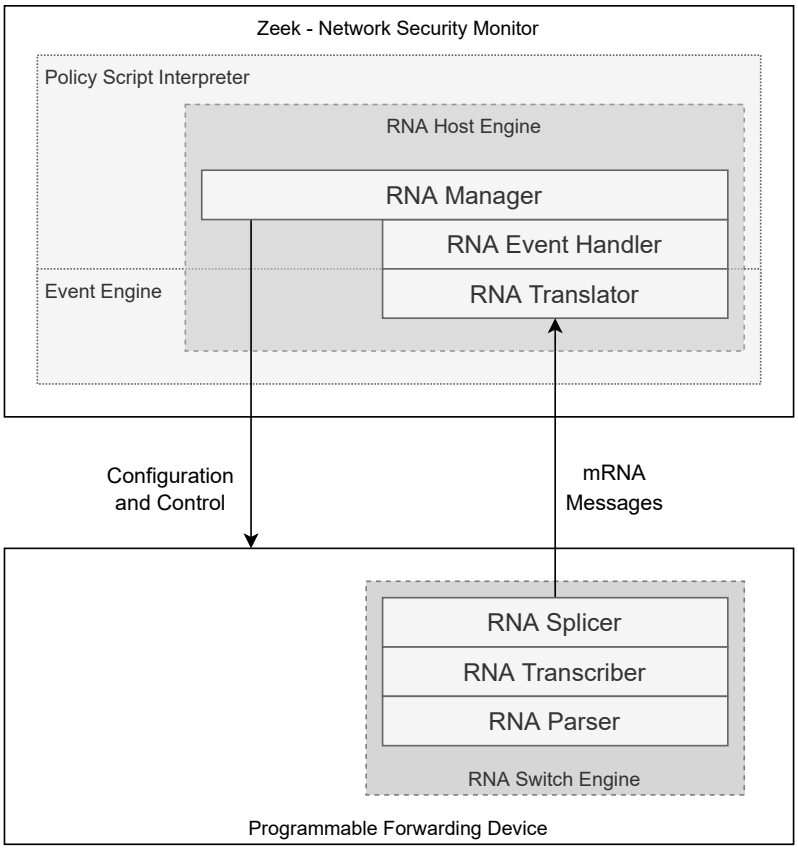


Source: Cordeiro, Marques and Gaspary (2017), adapted from Kim and Lee (2016).

3 RECONFIGURABLE NETWORK ANALYTICS

In this chapter, we present an extended version of the Reconfigurable Network Analytics (RNA) framework, an approach for offloading some of Zeek's operations discussed in Figure 2.2 to programmable forwarding devices compatible with P4. This proposal enhances the RNA framework proposed by Ilha (2022). Since our approach adds important capabilities to the original RNA, we describe the solution resulting from this incremental step, highlighting the new functionality.

Figure 3.1 – RNA Framework



Source: adapted from Ilha (2022)

Figure 3.1 illustrates the RNA framework. It consists of two high-level components: the RNA Host Engine, which executes in one of Zeek's worker nodes inside the IDS cluster, and the RNA Switch Engine, which is executed in a P4-programmable switch. Both components work together to offload packet analysis from Zeek to a programmable forwarding device. The RNA Switch Engine is able to parse packets and identify some of their characteristics, which are then summarized and sent to the RNA Host Engine. These summarized packets are called mRNA messages. When the IDS receives an mRNA mes-

sage, it is first processed by our Host Engine. It then converts the summarized message into Zeek's native structures, which can then be forwarded to Zeek's normal processing pipeline. This procedure allows us to bypass some operations that would be costly (such as protocol parsing and state tracking) and deliver this information closer or even directly to the Policy Script Interpreter (PSI, Figure 2.2) without disturbing Zeek's internal information flow. By doing so, we ensure no change is required on existing scripts running on the PSI.

In the original concept of RNA, depending on the IDS scripts chosen to be monitored by the operator, it is necessary to manually develop the additional software to be executed both by the Host Engine and the Switch Engine. This *ad-hoc* development process quickly becomes impractical when we increase the number of desired scripts to be offloaded. For this reason, we propose an automatic code generation mechanism, described later in Chapter 4, which allows RNA to be a modular solution, where varying combinations of Zeek scripts can be chosen to be monitored without the need to write new software by hand. Before detailing this automated process, we briefly review the operation of RNA components.

3.1 RNA Components in a Nutshell

Using Figure 3.1 as a reference, we present the high-level components of the RNA framework and their functionality. We start with the RNA Host Engine, since it is the managing part of the framework, and then we describe the RNA Switch Engine.

The Host Engine unfolds into three components. The RNA Manager is the controlling component of the deployment, which first configures the P4 switch, sets up a monitoring session, and loads all P4 code that is required to execute the offloaded tasks. After configuring the switch, now in the Host Engine, it registers into Zeek all RNA Translators (one per protocol of interest), so they receive mRNA messages. Translators are components responsible for waiting for such mRNA messages and translating them to Zeek native structures, and using those structures to trigger events, which are then consumed by the running scripts. The RNA Event Handler is another component that runs on the PSI and is initialized by the RNA Manager. It is designed as a debugging and logging component, capturing and handling events generated by the Translators.

The RNA Switch Engine is the program that executes in our P4-compatible programmable forwarding device. It has two components, and we will be following the route

of an incoming packet to explain them. The first component that processes a packet is the RNA Parser. It parses and extracts headers from each protocol, from the link layer, up to the application layer if required. After all the headers have been extracted, the packet enters P4's ingress pipeline, where the RNA Transcriber is executed. It extracts useful information from the packet and sets metadata that will later be used to build our summarized message while filtering some undesired packets. Having all the required metadata and going into P4's egress pipeline, the RNA Splicer builds and sends our summarized message, the mRNA, to the Host Engine with all information it may require to trigger a native Zeek Event.

Another important structure is the mRNA Message. It is a summary of a packet containing all the essential information that the Switch Engine extracted from it. Sending an mRNA message is more efficient than sending a whole packet because the original packet contains headers that would still need to be parsed, information that, in some cases, is not necessary and has not been validated. In the summarized message, all information from L2 up to the L7 layer is gathered, filtered, and, in some cases, even formatted according to Zeek's native structures, saving Zeek from doing these operations on its own. The information-gathering process still needs to happen, but it is offloaded to the Switch Engine, which runs in a purpose-built device, making it much more efficient for this task. So the more information the switch is able to extract, the less Zeek has to do.

In an ideal world scenario, we would like to extract all information that Zeek needs to trigger an event, but sometimes that is not possible. Zeek's internal structures track connection states and use detection heuristics, which, because of P4's limited processing power for general tasks, we are unable to implement. This requires the mRNA message to be modular, allowing us to send, together with it, the parts of the packet that could not be further processed in the switch. This ensures that P4 extracts all information it can, leaving the rest for Zeek to finish analyzing.

3.2 Detailed Framework Design

This proposed version of the RNA framework, differently from the original framework, goes further and specifies another level of subcomponents that make the framework adaptable to variable situations with varying sets of monitoring scripts. Before explaining the inner details of the framework, we will introduce two concepts that are fundamental for its understanding and will serve later as inputs for our code generator mechanism.

Those concepts are Offloader and Protocol Template:

- An *Offloader Template* is a set of files and settings that allows RNA to offload the monitoring of new protocols and the generation of corresponding events of interest. Like the idea of a module, it can be added and removed to the deployment without needing to develop new software or change existing components.
- A *Protocol Template* is a set of files and settings that allows RNA to parse a new protocol. An Offloader requires a set of Protocol Templates to be able to offload and parse the protocols that are of interest to Zeek scripts chosen by a network operator.

To explain the details of the architecture behind RNA, presented in Figure 3.2, we use the example of an incoming ICMPv4 Echo Request ("`ping`") packet and its trajectory through a deployed instance of RNA. In this hypothetical deployed system, the network operator also chooses other scripts that monitor protocols such as ARP, TCP, and UDP, which are not triggered by this specific incoming `ping` packet.

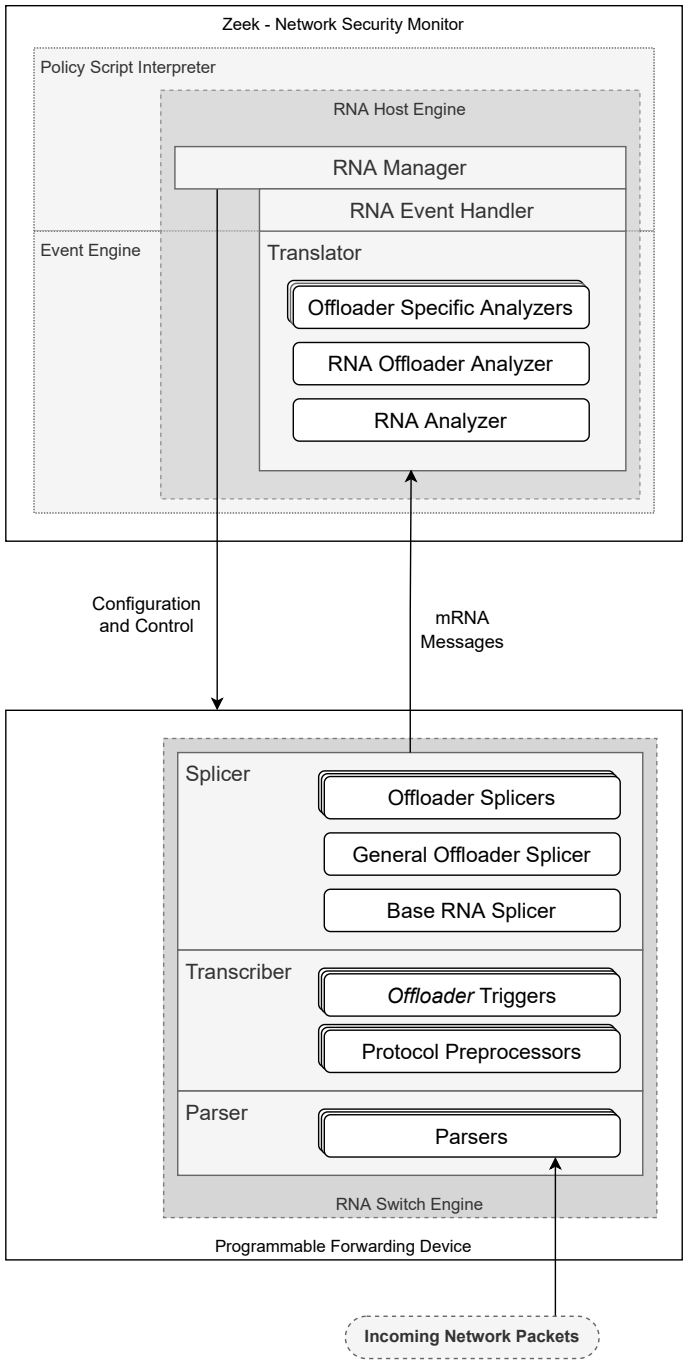
3.2.1 Switch Engine

The RNA Parser

As the packet arrives at the P4 switch, it is first processed by the programmable parser. The RNA Parser component is a state machine with all the parsers the Offloaders may need. Each one extracts protocol-specific header fields from the packet and forwards the payload to the next parser. Each protocol state and parsing instructions are provided by the Protocol Template of that specific protocol.

In our example, shown in Figure 3.3, the first parser is the Ethernet one, extracting its header, followed by the IPv4 and ICMP parsers. In the same figure, we also display other parser states that were not reached in this example, such as ARP, TCP, and UDP, each one provided by its own template. These unused parser states are displayed by dotted lines, exemplifying where they would be linked in the general parser. After the packet's headers are parsed, they are sent to the ingress pipeline, where the RNA Transcriber will process the incoming data.

Figure 3.2 – RNA Framework - Detailed view

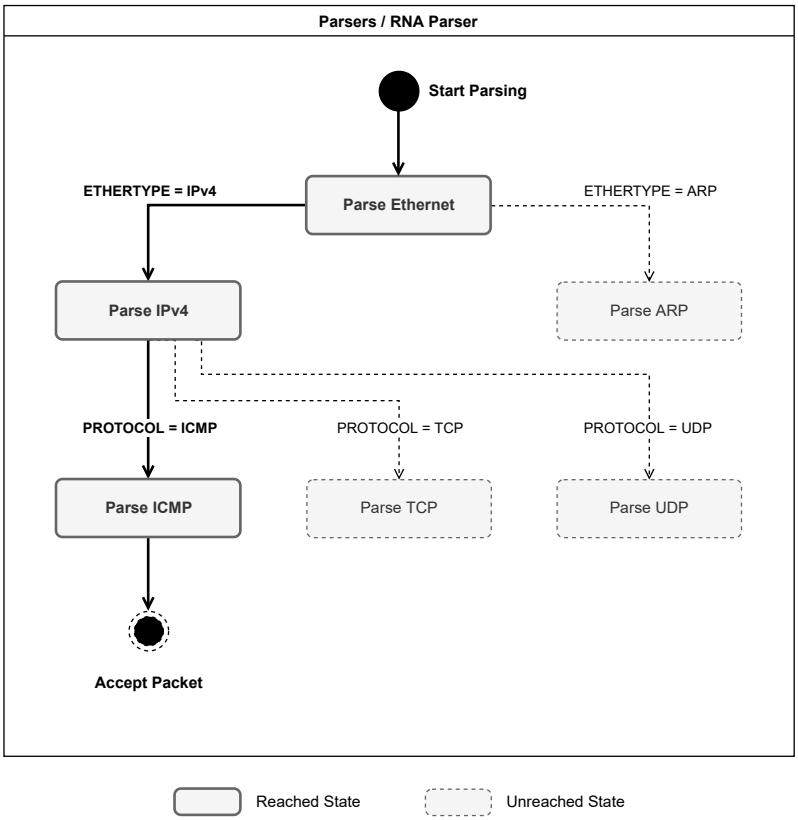


Source: the author (2022).

RNA Transcriber

In the RNA Transcriber, as shown in Figure 3.2, there are two types of subcomponents, namely, the Protocol Preprocessors and the Offloader Triggers, both of which may have multiple instances. We start explaining the Protocol Preprocessors since they are the first subcomponents to be executed inside this component.

Figure 3.3 – Parsing States - ICMP Parsing Example



Source: the author (2022).

The Protocol Preprocessors are functions that every Protocol Template may execute to extract protocol-related information from the packet and save it to the metadata structure. This subcomponent is optional, and every Protocol Template may have only one preprocessor. In our example, the preprocessor extracts the ICMP `type` field from the header and saves it to the metadata.

The next subcomponents to be executed in the Transcriber are the Offloader Triggers, which are conditions that identify which Offloader should be triggered. Each of these conditions is checked, and the first Offloader to have its trigger condition valid is marked as the triggered one in the packet's metadata. In our example of an ICMP Echo Request, our trigger condition is `icmp.type = 8`. This ensures the Offloader ICMP Echo Request is triggered when a `ping` packet has arrived.

After a valid Offloader is found, a copy of this packet is created. Using this cloned packet, the switch will be able to construct the mRNA message. The original packet will follow its flow and be delivered to the originally intended destination.

RNA Splicer

The RNA Splicer is executed after the packet leaves the ingress pipeline and it enters the egress pipeline[1]. The RNA Splicer is composed of three different types of Splicers: the RNA Base Splicer, the General Offloader Splicer, and the Offloader Splicers.

The RNA Base Splicer constructs the base header for the mRNA message, the first layer of the RNA protocol. This is a simple header that contains the RNA version and the RNA message type. We decided to use this simple header to allow for more expandability in the future, for example, adding debugging and other types of messages.

The next Splicer, The General Offloader Splicer, constructs a header that contains general information, which all Offloaders share, and is executed for every Offloader. This information includes, for example, source IP, destination IP, L3, and L4 protocols, and triggered Offloader. In our example of the ICMP Echo Request, the General Offloader Splicer sets both source and destination IPs, the L3 protocol to `IPv4`, the L4 protocol to `ICMP`, and the offloader type to ICMP Echo Request.

The third type of Splicer is the set of Offloader Specific Splicers. These are splicers that every Offloader has. Each Offloader Splicer constructs the header with the extra information required to execute its functionality, which was not yet present in the General Offloader Splicer. In our example, the ICMP Echo Request Splicer will construct a header with information about the ICMP `ping` packet: `icmp.type`, `icmp.code`, `icmp.sequence`, and `icmp.id`.

After every Splicer has finished constructing its headers, the mRNA message will be ready. The payload and headers, which together form the mRNA, are then merged and sent to Zeek's monitoring interface, effectively finishing the processing on the Switch Engine. From now on, Zeek's Translators will work to support the monitoring infrastructure.

3.2.2 Host Engine

In this section, we describe how the Host Engine receives the mRNA messages and uses them to trigger the offloaded events to which the monitoring scripts are subscribed. Figure 3.2 shows that the first components of the RNA Framework to receive the mRNA message are the Translators. Before any RNA Analyzer (Translator) receives

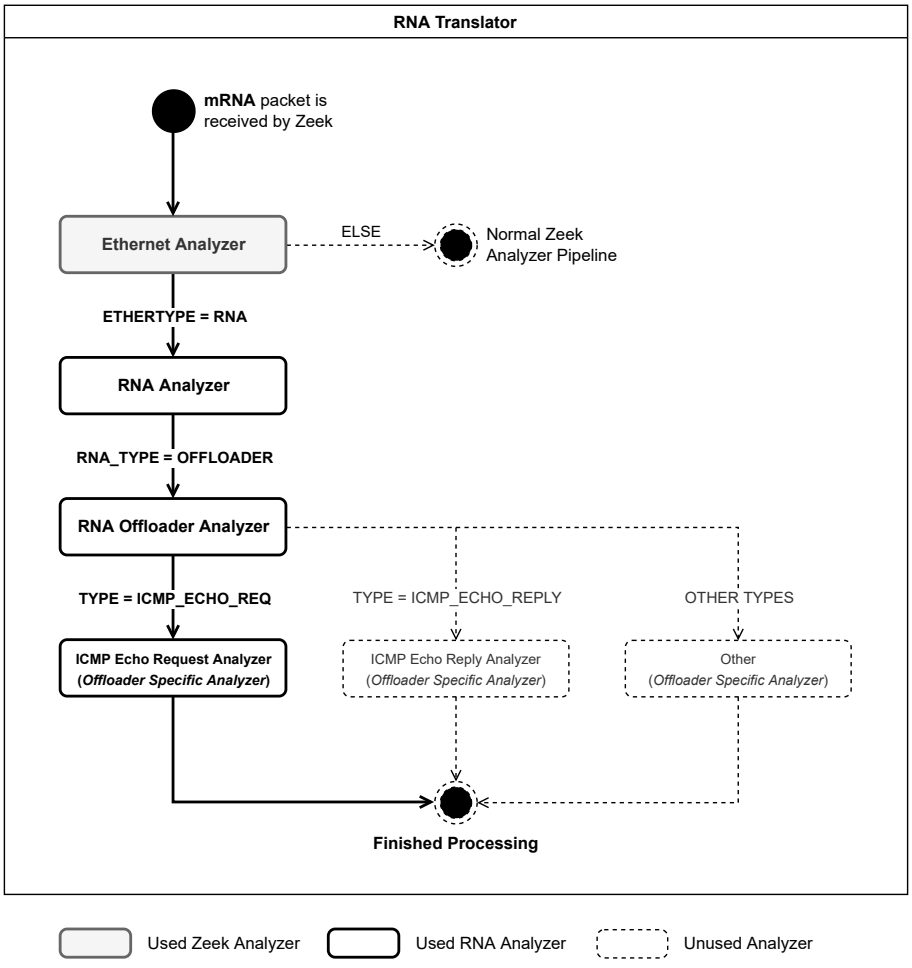
[1]Since we are explaining the architecture behind RNA, we are not considering the inner workings of a programmable data plane device, where a buffer connects the ingress and egress pipelines.

its packet, the packet is first received by Zeek's Ethernet Analyzer. When analyzing the packet, the previously registered mRNA ethertype code will ensure all mRNA messages are forwarded to our RNA Analyzer, as shown in Figure 3.4.

RNA Translator

The RNA Translator, similar to the RNA Splicer, is composed of three different types of Analyzers, which are subcomponents responsible for parsing each layer of the mRNA message. We use Figure 3.4 to explain how the analyzers are connected. First, it is important to note that Analyzers with a gray background are Zeek-provided, and Analyzers with a dashed border are Analyzers present in our deployment, but they are not invoked in our ICMP `ping` packet example (explained at the beginning of Section 3.2).

Figure 3.4 – Analyzers - ICMP Translation Example



Source: the author (2022).

The RNA Analyzer is the first Analyzer of the RNA architecture to receive the mRNA packet. As explained in the previous section, the first layer of the mRNA message format has a generic header that indicates only the protocol version and the message type. This is the header that is extracted by the first Analyzer. In the case of this project, the only message type we use is the Offloader type[2]. Using our previously described example of the ICMP `ping`, the resulting mRNA packet will have its type set to Offloader, forwarding it to the RNA Offloader Analyzer.

Next, the RNA Offloader Analyzer is the second Analyzer of our RNA implementation to receive the mRNA packet. It parses the header generated by the General Offloader Splicer with Offloader generic information. This Analyzer is also responsible for delivering the mRNA packet and its payload to the Offloader-specific Analyzer after extracting its layer's header by using the Offloader `type` code. In our example, the next Analyzer to be executed is the ICMP Echo Request Analyzer.

The last Analyzer to be executed is the Offloader Specific Analyzer. This is a subcomponent every Offloader must have. It is responsible for converting the received information to Zeek's native structures. It can then trigger one or more desired events, which will be consumed by the monitoring scripts. In our example, the ICMP Echo Request Analyzer is going to trigger Zeek's native `icmp_echo_request` event. By triggering this event, we ensure all scripts that subscribe to it will perform exactly as if Zeek had processed the ICMP `ping` request entirely in its own built-in infrastructure.

As previously explained in Section 3.1, an Analyzer directly triggering a Zeek Event is the ideal case, but not always possible. In the cases this is not possible, the Analyzer has the option to extract the mRNA headers and forward the rest of the payload to another Zeek-provided Analyzer. This other Analyzer, which is already part of Zeek's infrastructure, can then analyze the packet and trigger the proper events. This is also called the *fall-back mode* by Ilha (2022).

[2]There are actually three types of Offloaders, but for simplification purposes, in this description, we present the Offloader type as being one unique type.

4 AUTOMATIC CODE GENERATION FOR RNA

In this chapter, we present a mechanism for automatic code generation for the RNA Framework. This enables network operators to deploy the framework without having the experience and knowledge to develop software for Zeek or programmable forwarding devices (P4).

The code generation mechanism uses two sets of inputs: the Zeek scripts, whose events should be offloaded, and a pool of Protocol Templates and Offloader Templates (whose concepts we introduced in Section 3.2). These constructs are used as sources of templates and resources in order to implement the software required to offload the events subscribed by the desired scripts. We also propose the generation of a single Zeek Plugin package, which, when initiated, automatically deploys the code for both the Host Engine and the Switch Engine (instead of having two separate deployable packages).

4.1 Overview

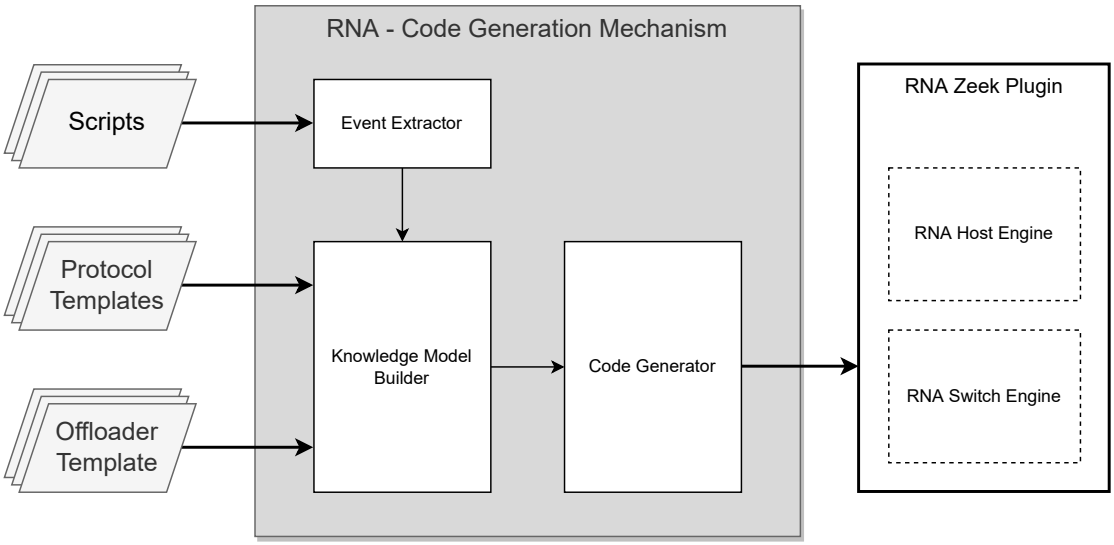
In this section, we present an overview of the RNA Code Generation Mechanism, starting with its inputs and expected output, which is illustrated in Figure 4.1. The main input for the mechanism is the set of Zeek scripts, whose monitoring the network operator is interested in. These scripts need to be provided in their entirety, so we can identify the events to which they subscribe.

The second set of inputs is what we call *templates*, which can be either Protocol Templates or Offloader Templates. They are a pool of known implementations of protocols and events that can be used to offload scripts. A template being present in this pool does not mean it will be included in the final output, but it means it is available in case its implementation is needed by a script.

The desired output of our mechanism is a single Zeek Plugin following the structure previously presented in Section 3.1 and illustrated in Figure 3.2. This Zeek Plugin, when executed, should: configure the switch by creating a mirroring session for the Zeek monitoring system and deploying the P4 code; and configure Zeek by registering all Translators in Zeek's Event Engine and loading the RNA Event Handler. This eliminates the need for the operator to coordinate the deployment of two separate systems, i.e., the RNA Host Engine and the RNA Switch Engine.

To be able to execute this task, the first objective of the code generation process

Figure 4.1 – RNA - Code Generation Mechanism



Source: the author (2022).

is to analyze all the provided Zeek scripts and identify which are the observed events in every script. Once this pool of events is known, we select Offloaders (from the templates pool) that are capable of offloading those events. This step is finished and succeeds if we find at least one Offloader for every event.

After all events and Offloaders have been selected, the mechanism must ensure all templates for the protocols required by these Offloaders are available. The Protocol Templates are required so the Offloaders can interpret the desired headers. After all this knowledge model is complete, the mechanism generates all required source files.

4.2 Detailed Mechanism Design

The operation of our RNA Code Generation Mechanism can be described in three different stages. The first stage identifies the events that our input scripts subscribe to. The second stage is building our knowledge model, which receives as inputs all Protocol Templates, Offloaders, and events of interest, that the network operator requested to be offloaded. We call this knowledge model *ProtocolGraph*. The last stage is the actual code merge and generation process using this structured and validated knowledge model from the previous step. We start explaining the first stage of our mechanism, i.e., extracting the subscribed events from the Zeek Scripts.

4.2.1 Event Extraction

To identify which events a Zeek Script subscribes to, we first need to parse the script as a whole. The mechanism does it using Zeek's provided grammatical definition. After parsing all the scripts, we search the Abstract Syntax Tree for event handlers, returning their event identifiers. Figure 4.2 illustrates a fragment of a Zeek Script that detects FTP Bruteforce attacks. For this particular example, the mechanism would identify the event handler declaration on line 20, which is the `ftp_reply` event.

Figure 4.2 – Section of a Zeek Script

```
1  ##! FTP brute-forcing detector, triggering when too many rejected usernames or
2  ##! failed passwords have occurred from a single address.
3
4  @load base/protocols/ftp
5  @load base/frameworks/sumstats
6
7  @load base/utils/time
8
9  module FTP;
10
11 export {
12     # Hidden to enhance readability
13 }
14
15
16 event zeek_init() {
17     # Hidden to enhance readability
18 }
19
20 event ftp_reply(c: connection, code: count, msg: string, cont_resp: bool) {
21   local cmd = c$ftp$cmdarg$cmd;
22   if ( cmd == "USER" || cmd == "PASS" ) {
23     if ( FTP::parse_ftp_reply_code(code)$x == 5 ) {
24       SumStats::observe("ftp.failed_auth", [$host=c$id$orig_h], [$str=cat(c$id$resp_h)])
             ;
25         }
26   }
27 }
```

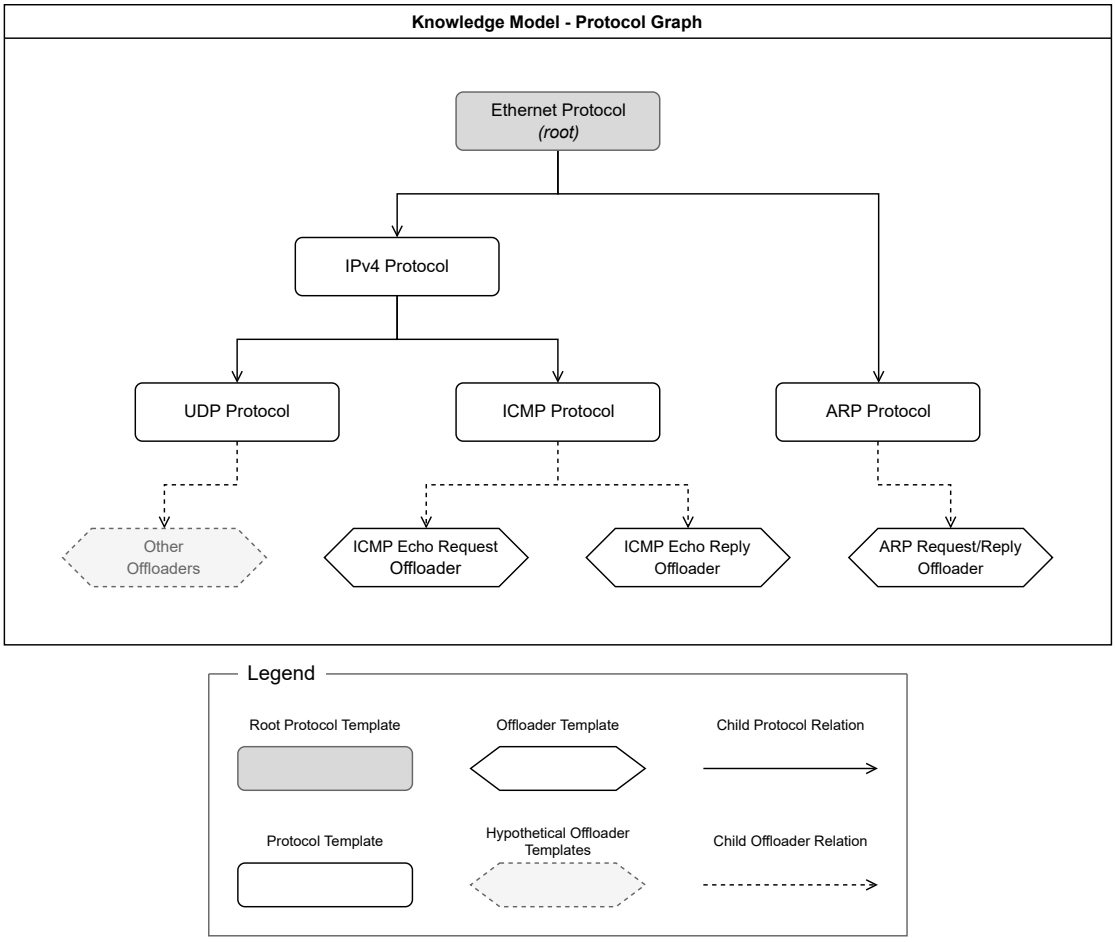
Source: the author (2022).

In this stage, the most important information to be forwarded to the next step is the identifiers of the events we want to offload. Those will indicate the requirements of our deployment. The name of the scripts are also passed to the knowledge model as a logging and debugging asset but are no longer essential for the final functionality.

4.2.2 Knowledge Model Builder

The core logic behind the implementation of our RNA Code Generation Mechanism relies on a structure we call *ProtocolGraph*. This is a graph structure that stores all protocols required for our events of interest. The Offloaders are linked to their final protocol layer in the graph, generating, in the end, a structure as presented in Figure 4.3. This structure is a graph, with a node we assign as root, in most cases, the ETHERNET protocol. The other protocols are linked to their respective parents. The Offloaders are linked to the protocol they analyze, which does not need to be a leaf protocol.

Figure 4.3 – Knowledge Model - Protocol Graph



Source: the author (2022).

The procedure to generate and validate the *ProtocolGraph* is outlined in Algorithm 1. The pseudo-code will be used to describe and guide the explanation of our method, and we will refer to the lines according to their respective roles.

Algorithm 1: Knowledge Model Build Algorithm

Input: $templates$, $events$, $forced\_offloaders$
Output: $graph$

1 VALIDATECOMPONENTLIST($templates$)

2 $protocols \leftarrow$ FilterListByType($templates$, $ProtocolTemplate$)
3 $offloaders \leftarrow$ FILTERLISTBYTYPE($templates$, $OffloaderTemplate$)

4 $offloaders \leftarrow$ REQOFFLOADERS($offloaders$, $events$, $forced\_offloaders$)

5 **if** $offloaders$ *is empty* **then**
6 ⌊ **raise** *exception* ▷ nothing to be offloaded

7 $root \leftarrow$ FINDROOTPROTOCOL($protocols$)
8 $graph \leftarrow$ LINKGRAPH($root$, $protocols$)

9 **if** HASCYCLES*(graph)* **then**
10 ⌊ **raise** *exception* ▷ protocol graph has cycles

11 $protocols \leftarrow$ REMOVEUNREACHABLEPROTOCOLS($graph$, $protocols$)
12 ATTACHOFFLOADERS($graph, offloaders$)

13 TRIMUNUSEDPROTOCOLS($graph$)

14 SETPROTOCOLDEPTHS($graph$)
15 SORTPROTOCOLS($graph$)
16 SORTOFFLOADERS($graph$)
17 SETOFFLOADERSUIDS($graph$)

18 **return** $graph$ ▷ successful graph generation

Template loading and filtering

As the mechanism is executed, the first step is to load all templates and validate their versions, files, and requirements. All the templates are loaded and stored in a list, even if they are not used later in the final model. After they are loaded, we separate them between Protocol Templates and Offloader Templates (Algorithm 1, lines 1-3).

After all offloader templates are validated, we select what offloaders are required (line 4). This selection process is based on two parameters: the event identifiers (provided by the Event Extraction stage) and a list of "forced" offloaders (provided by the network operator when invoking the code generator mechanism). The first offloaders to be assigned as required are the ones explicitly requested by the network operator. Next, we select an offloader for each event, ensuring all events are covered. If, at the end of

this process, one or more events do not have offloaders associated with them, the process fails and is aborted. Otherwise, we proceed. We then check if the list of Offloaders is not empty and proceed to the next section (lines 5-6).

Graph building

After the templates have been validated and the desired Offloaders have been selected, the mechanism needs to find the root protocol. If none or more than one root Protocol Template is found, an exception is raised, and the process is terminated (line 7).

Each Protocol Template has its own identifier, which is a unique string, as well as the identifiers of the parent protocols. Using these identifiers, the Protocol Templates are linked to their parent, making sure the links are all valid (line 8), for example, the ICMP protocol is linked to its parent protocol, the IPv4. After linking, the algorithm validates there are no cycles on the protocol graph (lines 9-10). The reason the protocol graph can not have cycles is due to limitations on the algorithm's P4 implementation, where protocol headers can not be parsed twice for a single packet. By ensuring no cycles are present in the *ProtocolGraph*, we ensure protocol headers can only be used once. After the graph is built, the algorithm removes all unused protocols from the protocol list (line 11).

The algorithm links all the required Offloaders to their respective protocols (line 12). If any Offloader fails to be attached to its protocol, the algorithm aborts the execution. At this point, the *ProtocolGraph* is fully built, but it may have unnecessary protocols still attached to it. These protocols that have no Offloaders attached to them or any Offloaders attached to their child protocols are removed on line 13. It might sound strange that we build the graph to then trim and optimize it. Since the development of this mechanism was incremental, this is how it was originally developed, but we acknowledge it could be further optimized.

Sorting and Prioritization

After the *ProtocolGraph* is fully built, the mechanism sets the last metadata that will be needed in the code generation stage. The first metadata to be set is the *protocol depth*. For each protocol, the mechanism sets protocol depth as the maximum distance of that protocol to the root protocol (line 14). The depth is used to then sort the protocols and their Offloaders in a list (lines 15 and 16), which will later be consumed by the code generator. A list based on Offloader priority is also created at this time. The last step is

to set the unique identifier (UID) for the Offloaders (line 17). This ensures the Switch Engine and the Host Engine have the same identifier referring to the same Offloaders.

4.2.3 Code Generation

The code generation mechanism is based on a *master template* with blank slots, which are filled with the generated code. The code that fills the slots of the master template is either fully generated by the mechanism or is pulled from the protocol and Offloader templates.

During the design of the code generation mechanism, we decided between creating a structure similar to an AST (Abstract Syntax Tree) and fully generating the required code. The chosen solution was to use a combination of template files, with AST-like code generation. This ensures flexibility and ease of development, allowing the structure of the generated files to be quickly changed, as well as the generated code.

Next, we first describe how the code is generated for the P4 switch, and then we explain how it is generated for the Zeek Plugin. This code generation process could be, in most parts, parallelized since it is based on the previously generated *ProtocolGraph*, but we did not see the necessity to implement the parallelism at this time.

Code Generation for the Programmable Switch (P4)

The P4 code is mainly composed of three files, which are *headers.p4*, *parser.p4*, and *main.p4*. We explain this structure in a bottom-up approach, starting with the headers file. The headers file, as the name suggests, contains all the headers and data structures required by the output program. It is generated by merging all headers provided by the templates, which include protocol headers, mRNA headers, and Offloader specific headers[1], and others.

The *parser.p4* file contains all the information required for parsing and deparsing protocols. The parser, as described previously in Section 3.2, is a state machine. The states of the parser are all generated based on the information provided by the protocol templates. It uses the next protocol selector field to select one of the child protocols. Using our example from Section 3.2, shown in Figure 3.3, the Ethernet parsing state would use the `ethernet.ethertype` field to select either the *IPv4* or *ARP* protocols.

[1]The Offloader specific headers are headers created by the Offloader Splicer, which were explained in Section 3.2.

When parsing our example ICMP ECHO REQUEST packet, the value of the ethertype field would point to IPv4. Likewise, the value of the `Protocol` field in the IPv4 datagram would point to ICMP.

To support protocols with variable-size headers, the template has the option to use a custom parsing code, which replaces the default header extractor. Instead of extracting a fixed-size header based on a (fixed-size) structure, this custom code allows the templates to determine the received header size and parse the right amount of bits from the packet.

The main P4 file is the entry point for our P4 program. It defines the ingress and egress pipeline control structures while also loading the headers and the parser files. When generating the ingress pipeline, we split it into two generated sections, the *protocol preprocessors*, and the *Offloader triggers*. The protocol preprocessors section is generated by merging the preprocessors provided by every protocol template, which were explained in Section 3.2. The preprocessors are each wrapped with verification to ensure the protocol is valid for that packet, and are generated in bottom-up order. Figure 4.4 shows an example of a section of the ingress pipeline. The generated protocol preprocessors are displayed on lines 3 - 16, where each of the preprocessors is wrapped with an *if statement* to ensure the protocol is valid before executing the preprocessor.

Still in the ingress pipeline, the Offloader triggers are conditions provided with every Offloader template and are also wrapped with a test for protocol validity. They are sorted based on the Offloader's priority since only one Offloader may be triggered per packet. This is an if statement, which, when met, sets the `metadata.offloader` field to the Offloader's UID, which is defined in the *ProtocolGraph*. Using Figure 4.4 again as an example, on lines 20 - 39 the Offloader triggers are defined. The trigger condition for the *NTP Message Offloader* is seen on line 26. If this condition is true, the P4 switch will set the `meta.offloader_type` field to `RNA_NTP_MESSAGE_UID` (line 27).

Still in the *main.p4* file, the egress pipeline is composed of one generated section, the *Offloader splicers*. This code section is composed of *if* statements verifying the metadata's Offloader identifier. In the *if* body, the mechanism merges the splicer code, which is provided by every Offloader template and will be executed if the Offloader UID matches the one in the metadata. Figure 4.5 shows an example of the splicer section of the egress pipeline. Lines 2 and 6 verify the triggered offloader, and lines 3 - 5 and 7 - 9 are the splicer code for the *FTP Message Offloader* and the *NTP Message Offloader*, respectively.

Figure 4.4 – Section of the P4 Ingress Pipeline

```
// Hidden to enhance readability

if (hdr.ethernet.isValid()) {
    meta.protocol_l3 = hdr.ethernet.ethertype;
}
if (hdr.ipv4.isValid()) {
    meta.protocol_l4 = hdr.ipv4.protocol;
}
if (hdr.tcp.isValid()) {
    meta.src_port = hdr.tcp.src_port;
    meta.dst_port = hdr.tcp.dst_port;
}
if (hdr.udp.isValid()) {
    meta.src_port = hdr.udp.src_port;
    meta.dst_port = hdr.udp.dst_port;
}

// Hidden to enhance readability

if (hdr.tcp.isValid()) {
    if(hdr.tcp.flags.PSH == 1 && (hdr.tcp.src_port == 21 || hdr.tcp.dst_port == 21)) {
        meta.offloader_type = RNA_FTP_MESSAGE_UID;
    }
}
if (hdr.udp.isValid()) {
    if(hdr.udp.src_port == NTP_PORT || hdr.udp.dst_port == NTP_PORT) {
        meta.offloader_type = RNA_NTP_MESSAGE_UID;
    }
}
if (hdr.icmp_echo.isValid()) {
    if(hdr.icmp.type_ == ICMP_ECHOREPLY || hdr.icmp.type_ == ICMP_ECHO) {
        meta.offloader_type = RNA_ICMP_ECHO_MESSAGE_UID;
    }
}
if (hdr.icmp_ipv4_context.isValid()) {
    if(hdr.icmp.type_ == ICMP_DEST_UNREACH || hdr.icmp.type_ == ICMP_TIME_EXCEEDED) {
        meta.offloader_type = RNA_ICMP_CONTEXT_MSG_UID;
    }
}
```

Source: the author (2022).

Code Generation for the Zeek Plugin package

The generation of the Zeek Plugin is much simpler than the P4 code generation. Most of the Zeek Plugin is composed of static files, which are copied from the *master template* and from the templates. The rest of the generation consists of registering the template's Analyzers, defining constants, and creating *read me* and *version* files.

The most important files to be generated are the main plugin file (`Plugin.cc`), the main Zeek Script file (`main.zeek`), and the building rules (`CMakeLists.txt`). The main plugin file needs to include the C++ header files and register the Analyzers. The main Zeek Script file needs to contain instructions to load the Analyzers and bind them to the specific Offloader UIDs, which were defined by the *ProtocolGraph*. To ensure all the template's code files are properly compiled, the mechanism needs to add all their

Figure 4.5 – Section of the P4 Egress Pipeline

```
// CONSTRUCT OFFLOADER SPECIFIC HEADERS
if (meta.offloader_type == RNA_FTP_MESSAGE_UID) {
    hdr.ftp_message.setValid();

    // Hidden for clarity, splicer continues here
} else if (meta.offloader_type == RNA_NTP_MESSAGE_UID) {
    hdr.ntp_message.setValid();

    // Hidden for clarity, splicer continues here
}
```

Source: the author (2022).

file paths to the *CmakeLists* recipe. The last step in the composition of the Zeek Plugin package is to generate the `README` file, which describes what the plugin does, and the `VERSION` file, which contains the version of the generated plugin.

The proposed mechanism was conceived to embed the P4 code in the Zeek Plugin package so that when the RNA Manager is initiated, the P4 code can be deployed to the programmable forwarding device. This was not implemented in the prototype, but we briefly describe how it could be implemented. This could be implemented by first generating the P4 code, then compiling it, either with a P4 command or even in a Docker container, to ensure all dependencies are met. After that, when generating the Zeek Plugin package, the mechanism would embed the output of the P4 code compilation (a *JSON* file) and generate a function to load it to the switch when the RNA Manager was initialized.

5 PROOF OF CONCEPT AND EVALUATION

In this chapter, we present how the automatic code generation mechanism proposed in Chapter 4 was implemented in a fully functional proof of concept. We also describe how we evaluated the gains in performance and ease of development and deployment for users of our framework.

To facilitate the development of this project and to keep the scope within a defined limit, we did not use P4-compatible hardware. Instead, we used the Behavioral Model version 2 (BMv2) software-based switch (The P4 Language Consortium, 2020) to run the P4 code. This enabled us to develop with high agility the generation tool and to test the development more frequently, making sure the prototype was functional.

5.1 Prototype Implementation and Deployment

In this section, we describe some implementation details of the code generation mechanism that were not previously discussed. We divide this explanation into three subsections. We first explain the structure of the Protocol and Offloader Templates, which are two of the main inputs for our mechanism. Then we explain some of the implementation aspects of our prototype. To finalize, we explain how the output of our mechanism, the automatically generated code, is deployed in a virtualized network with an emulated P4 switch.

5.1.1 Protocol and Offloader Templates

The Protocol and Offloader Templates have a similar structure based on a configuration file. This configuration file is in the Hjson (HJSON. . . , 2022) format, which is based on the well-known JSON format. We first explain the Protocol Templates using the example configuration shown in Figure 5.1. Each Protocol Template needs to provide header definitions so it may be parsed. This is done by providing the name of the header structure and the file where it was defined (lines 13 and 12). The protocols may optionally have a custom *ingress processor* and a custom parser to enable parsing of variable-size headers, both of which were explained in Section 3.2.

To enable a Protocol Template to be linked to its children, we need to define the

Figure 5.1 – Template configuration file: `icmp.hjson`

```
1  {
2      "zpo_type": "PROTOCOL",
3      "zpo_version": "0.0.1",
4      "id": "icmp",
5      "parent_protocols": [
6          {
7              "id": "ipv4",
8              "id_for_parent_protocol": 1 // DECIMAL id to identify this protocol in the
                   parent protocol
9          }
10     ],
11     "header": {
12         "header_file": "icmp_header.p4",
13         "header_struct": "icmp_h"
14     },
15     "next_protocol_selector": "type_", // A field of the header template provided
16     "ingress_processor": "ingress_processor.p4" // Optional
17 }
```

Source: the author (2022).

parameter called `next_protocol_selector`. It specifies the field of the protocol header that will be used to select the next protocol (line 15). To link the protocol to its parent, we need to specify the parent protocol identifier (line 7) and specify what value the `next_protocol_selector` parameter must have for the packet to be forwarded to the child protocol (line 8). With this structure, the mechanism is able to generate all required code for parsing the protocols in the Switch Engine.

An Offloader Template is also based on a configuration file, and we use Figure 5.2 as an example. Each Offloader needs to be associated with a Protocol Template by its identifier (line 5). Each Offloader then must have a header structure definition (line 8) for its mRNA message. This header structure is defined in a P4 file, which is also a part of the template, and its path must be specified in the configuration (line 9). The rest of the parameters used for the Switch Engine are extracted from separate P4 files, the splicer, and the trigger condition (lines 10 and 11). For the Host Engine, the template must specify the C++ code and header files, as well as the name and *namespace* of the Analyzer (lines 14 to 22). To finalize, the configuration also specifies what Zeek Events the Offloader is capable of offloading (lines 23 to 26).

5.1.2 Prototype Implementation

Our prototype implementation of the code generation mechanism follows all architectural details explained in Chapters 3 and 4. It was implemented in Python 3 in a

Figure 5.2 – Template configuration file: `icmp_echo_message.hjson`

```
{
    "zpo_type": "OFFLOADER",
    "zpo_version": "0.0.1",
    "id": "icmp_echo_message",
    "protocol": "icmp_echo",
    "is_ip_based": true,
    "p4": {
        "header_struct_name": "icmp_echo_message_h",
        "header_file": "icmp_echo_message_header.p4",
        "splicer_file": "constructor.p4",
        "trigger_file": "identifier.p4"
    },
    "zeek": {
        "analyzer_namespace": "zeek::packet_analysis::BR_UFRGS_INF::RNA::ICMP",
        "analyzer_class": "RnaIcmpEchoAnalyzer",
        "analyzer_id": "RNA_ICMP_ECHO",
        "header_files": [
            "RnaIcmpEchoAnalyzer.h"
        ],
        "cc_files": [
            "RnaIcmpEchoAnalyzer.cc"
        ],
        "offloaded_event_ids": [
            "icmp_echo_request",
            "icmp_echo_reply"
        ]
    }
}
```

Source: the author (2022).

modular way so it could be maintained and further developed as the RNA Framework grows. To explain further details of the implementation that were not yet discussed, we follow the same structure used to explain the mechanism details in Section 4.2, and we start with the Event Extraction part.

The Event Extraction component was a strong reason for choosing Python as our programming language since Zeek provides its own Python library for parsing Zeek Scripts. In this component, we parse the provided Zeek Scripts and search their Abstract Syntax Tree (AST) for event handler declarations. Once those handler declarations are found, we extract their identifiers and forward this list of identifiers to the next component, the Knowledge Model Builder.

The Knowledge Model Builder uses as inputs the templates and the Zeek Script events. In this component, we create a graph structure following Algorithm 1 and the procedures explained in Section 4.2. In our implementation, we use exceptions to handle the flow of the algorithm and abort when any requirements are not met. Since our implementation of the Knowledge Model Builder does not differ from the algorithm explained in Section 4.2, we do not repeat the explanation in this section.

The Code Generation component in our prototype uses template files with markers to insert the generated code in the correct location. The files used for this purpose are called *master template* files, and this is what defines the structure and organization of the output of our mechanism. In the *master template*, markers are predefined strings in specific formats that indicate where a specific code section will be inserted. To generate code that will replace these markers, we use a structure similar to an AST, where each code element is a node, implemented using a class containing its children nodes. The mechanism first builds this structure, linking all nodes, then converts the root node to a string. This conversion is done recursively for each node, returning, in the end, the complete generated code. When the code is generated, we replace the corresponding marker with the generated code and save the file to the output directory.

The output of our mechanism is also composed of code that is provided with each template. To merge these provided sections of code, we use the same strategy as explained in the previous paragraph. We use template files with markers to define where each part of the code will be inserted. We also split some files, mainly on the Zeek Script, as *no-edit* files. These *no-edit* files are copied to the output location unaltered because they do not need any modifications, and some of them are static files required by the Zeek Package structure.

When our mechanism is executed, it generates an output folder containing all the automatically generated code. As explained in the previous chapter, we have not yet implemented the deployment of the P4 code using the Zeek Package, so we split this output folder into two sub-folders. One of the folders contains the P4 code for the switch, and one contains the Zeek Plugin package. In the next section, we explain how the P4 code and the Zeek Plugin are deployed.

5.1.3 RNA Deployment

The deployment of the RNA framework takes place in a virtualized network and uses an emulated P4 switch, so no specific hardware or Programmable Forwarding Devices (PFDs) are required to test our approach. To emulate the switch, we use the *p4app* tool, which sets up a virtualized network and instantiates a BMv2 switch (The P4 Language Consortium, 2020) within a Docker container. The *p4app* tool (The P4 Language Consortium, 2019) compiles the P4 code and loads it into an emulated programmable forwarding device. To set up the network, *p4app* uses a tool called *mininet* (LANTZ;

HELLER; MCKEOWN, 2010), which creates all the interfaces for each of our devices (hosts and switches) and allows us to simulate different network topologies. To run Zeek, we use a custom Docker image that contains all required dependencies. When executing Zeek, we link this Docker container's network to the *p4app* container network, which allows us to run Zeek on any interface of the virtual switch, but, usually, on the port setup as a mirroring port.

In our tests, we used a simple topology with two hosts linked by an emulated P4 switch, with a mirroring port where Zeek is listening. The mRNA messages generated by the switch are sent to the mirroring port, where Zeek is listening for incoming packets. To generate the traffic that is analyzed by our mechanism, we used two different methods. The first method is using *p4app* to open terminals in virtual hosts, where we are able to run programs and generate traffic for the framework to process. The second method is using packet traces that were previously captured and forwarding these traces to be processed as incoming traffic.

5.2 Evaluation

We now present the evaluation of our proposed approach, both the RNA framework and the automatic code generation mechanism. We assess the ability of our code generator to generate correct code and how it enables an inexperienced network operator to offload monitoring scripts to PDPs. Last, we assess the performance of the output of our solution, the automatically generated instance of the RNA framework. These aspects can be formalized as the following research questions (RQs):

- *RQ1*: Is the code generator mechanism able to correctly generate code to offload a set of Zeek Scripts using RNA?
- *RQ2*: How many lines of code did the code generator yield? And how many extra lines would a developer or network operator need to code to deploy the solution?
- *RQ3*: How does the performance of a Zeek deployment with RNA compare to a deployment without RNA?

To answer those questions, we deploy an automatically generated instance of RNA and test it using traces containing attacks that trigger warnings on a set of predefined scripts. To limit the scope of this project, we decided that the supported scripts should *(1) not require any state management by the generated RNA code* and should *(2) not require*

any stream reassembly. Those scripts are:

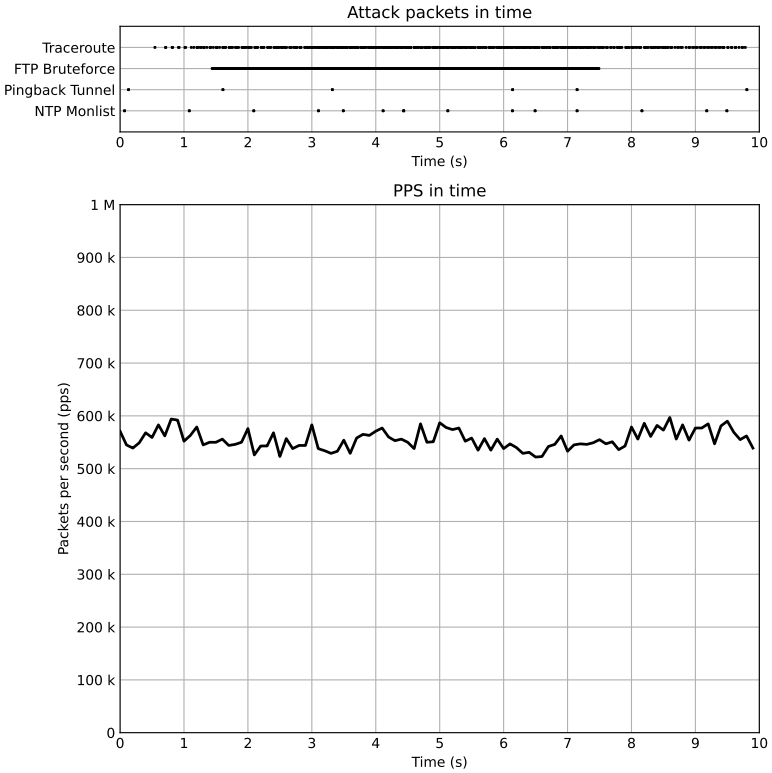
- *FTP Bruteforcing*: This script detects FTP authentication brute-force attacks. It triggers a warning after a number of unsuccessful login attempts by an FTP client.
- *Detect traceroute*: Detects trace-route attempts by monitoring *ICMP Time Exceeded* messages. This script is provided by Zeek, and originally it uses *Signature Detection* (The Zeek Project, 2022a). Since we do not support this feature on RNA, we disabled the usage of *signatures* on this script.
- *ICMP Pingback*: Developed by Reardon (2021), it detects the usage of ICMP ping tunnels created by the Pingback C2 tool.
- *NTP Monlist*: This script detects NTP Monlist attacks (NTP Monlist Detection, 2021).

5.2.1 Experiment Workload and Dataset

The workload used for our experiments was a combination of a legitimate dataset with an attack dataset, some of which were generated by us. The legitimate dataset used was the *CAIDA Anonymized Internet Traces 2016* (CAIDA, 2016), which comes from a high throughput backbone. Since this packet trace was too big, we selected only a small (but dense) ten-second window to use for our experiments, which we now refer to as our *legitimate dataset*. This packet trace was then merged with smaller well-known attack traces, whose combination we call *attacks dataset*, with 1200 packets. Combining these two traces ensures our selected scripts trigger warnings. This combined dataset is the one used for our experiments, which we refer to as the *combined dataset*. The workload has 5.5 million packets, a mean of 556 thousand packets per second (kpps) and 3269 megabits per second (Mbps). Figure 5.3 shows the variation of packets per second (pps) for our dataset, as well as the placement of the attacks used.

Merging different datasets, especially in our case, where one contains all malicious packets, may raise concerns regarding the detection being facilitated due to, for example, IP address and port distribution. In our case, this is not a problem since none of the monitoring scripts used work by analyzing traffic patterns or using statistical methods. All monitoring scripts analyze packets for specific signals and behaviors of interest. Therefore, merging those two datasets does not alter the detection ratio nor facilitates the detection for the used scripts.

Figure 5.3 – Packets Per Second (pps) for the dataset



Source: the author (2022).

5.2.2 Experiment setup and methodology

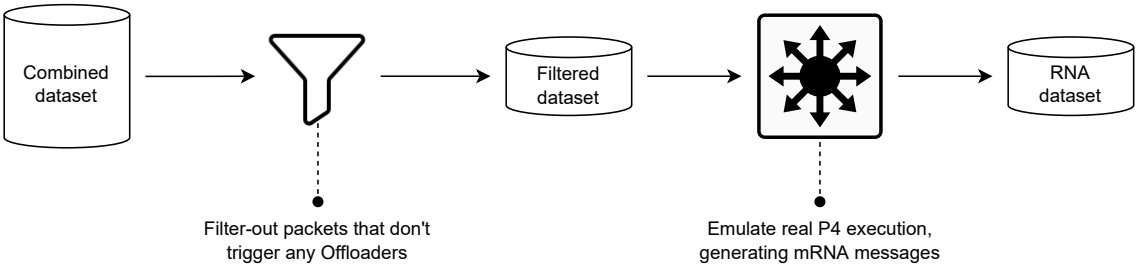
In this section, we describe the setup of our experiment. The experiment used the same technologies described in Section 5.1.3, relying on network virtualization, P4 emulation to run our switch, and containerization to run Zeek. The objective of our evaluation was to compare the functionality and performance of the Zeek Scripts without RNA compared with RNA.

Switch emulation does not perform as fast as a real P4 hardware switch, which makes it impossible for us to execute our experiment with a P4 switch in real-time since we only use emulated switches. In this scenario, the emulated switch would become a bottleneck, preventing the traffic from reaching Zeek at the same rate as it enters the P4 pipeline. For this reason, we assess only the performance of the Zeek monitoring system. We assume that a (hardware) programmable forwarding device would be able to execute the program at line rate if the provided program fits the device's memory.

To overcome the performance deficit of emulated P4 switches, we process the combined dataset before running the experiment, effectively creating a second dataset. This second dataset represents a real-world output of a P4-switch, receiving our combined dataset and processing it at line rate. We call this second dataset the *RNA dataset*.

To explain the generation of this second dataset, we use Figure 5.4. The first step is to select from our dataset only the packets that may trigger an Offloader, which will eventually generate an mRNA message. With this intermediate trace, we execute our emulated P4 switch, which is now able to process the dataset faster due to the decreased amount of traffic. This results in a dataset with only mRNA messages, the *RNA dataset*. It is also important to note that in all datasets during this process, all packets have their timestamps preserved, and real behavior is emulated.

Figure 5.4 – RNA dataset creation diagram



Source: the author (2022).

Now that we have our two datasets, the *combined dataset* and the *RNA dataset*, we need to execute Zeek in both scenarios and compare its output and performance. This is done by running Zeek in a Docker container connected to the host computer by a virtual network interface. In this network interface, we replay those two traces, one at a time, and record the memory and CPU usage. Each scenario was executed fifteen times to account for variability. The experiments were executed on a notebook with an Intel Core i9-10885H ($5.3\,\mathrm{GHz}$, 8 cores, 16 threads) CPU, $16\,\mathrm{GB}$ ($2{\times}8\,\mathrm{GB}$) of DDR4 RAM ($3200\,\mathrm{MT/s}$), and a $1\,\mathrm{TB}$ NVMe SSD. While the experiments were executed, in order not to affect the results, no other non-essential services were executed on the computer.

5.2.3 Results

In this section, we describe the results of the experiment and functional assessment of our code generation mechanism. We first present the functional results, answering

research questions one (RQ1) and two (RQ2). We later present the performance results, answering *RQ3*.

Functional results

To assess whether our proposed mechanism works, we used our prototype and the attacks dataset to compare the output of a Zeek deployment with and without RNA. After executing both setups and comparing the output of Zeek's *notices*, we concluded that our code generation mechanism was able to generate code to successfully offload four different scripts, namely *FTP Bruteforcing*, *Detect traceroute*, *ICMP Pingback*, and *NTP Monlist*. All of these scripts were able to detect, with complete accuracy, all the attacks present in the workload of the experiment, resulting in no difference between the execution with and without RNA in terms of detection.

To answer the second question, we manually inspect the generated code for RNA. Our objective is to check whether our approach is helpful and facilitates the deployment of RNA. Table 5.1 summarizes the main results obtained. The generated output instance of RNA for offloading our four Zeek Scripts (described above in this section) has 2967 lines of code. To develop a new Protocol Template, according to Table 5.2, a median of 40 lines of code would need to be written, and for a new Offloader Template (Table 5.3), a median of 224. This gives developers a big advantage over writing a fully standalone solution since templates are small and easier to maintain than a complete solution. The main advantage leans on the reuse of Protocol and Offloader Templates. Using templates, a network operator is able to deploy RNA without writing a single line of code, only with a command. The automatic code generator identifies the needed events to offload the desired scripts and generates the complete code.

Table 5.1 – Lines of Code per Script Count

Monitoring Script Count	**Generated Lines Median**
1 Monitoring Script	2113.5
2 Monitoring Scripts	2421.5
3 Monitoring Scripts	2714.0
4 Monitoring Scripts	2967.0

Source: the author (2022).

Table 5.2 – Lines of code of Protocol Templates

Protocol Template	**Lines of Configuration**	**Lines of code**	**Total Lines**
Ethernet Protocol	13	13	26
IPv4 Protocol	17	26	43
IPv6 Protocol	17	20	37
ICMP Protocol[1]	59	77	136
TCP Protocol	22	45	67
UDP Protocol	21	10	31
Total	*149*	*191*	*340*
Median	*19*	*23*	*40*

Source: the author (2022).

Table 5.3 – Lines of code of Offloader Templates

Offloader Template	**Lines of Configuration**	**Lines of code**	**Total Lines**
NTP Message	27	152	179
ICMP Echo Message	28	157	185
ICMP Time Exceeded	28	305	333
FTP Request and Reply	28	235	263
Total	*111*	*849*	*960*
Median	*28*	*196*	*224*

Source: the author (2022).

Performance results

To answer our third research question and assess the performance gain of offloading scripts with RNA, we replayed both of our datasets, the *combined dataset*, and the *RNA dataset*. When replaying the RNA dataset, we enabled our Zeek Plugin, which processed the incoming mRNA messages. This resulted in a significant gain in performance. As Figure 5.5 shows, the mean CPU[2] usage without RNA is 109%, and memory usage reaches a maximum of 960.64 MB by the end of the experiment. With RNA (simulated by using the *RNA dataset*), mean CPU utilization was 1.9% and the maximum memory usage was 235.07 MB. Additionally, without RNA, a mean of 35.24% of the packets was dropped. The high dropped packet rate resulted in one of the attacks, the *FTP Bruteforce Attack*, not being detected in 93.3% of the iterations executed without RNA. Using our approach, no packets were dropped, and all attacks were detected in all executions of the experiment.

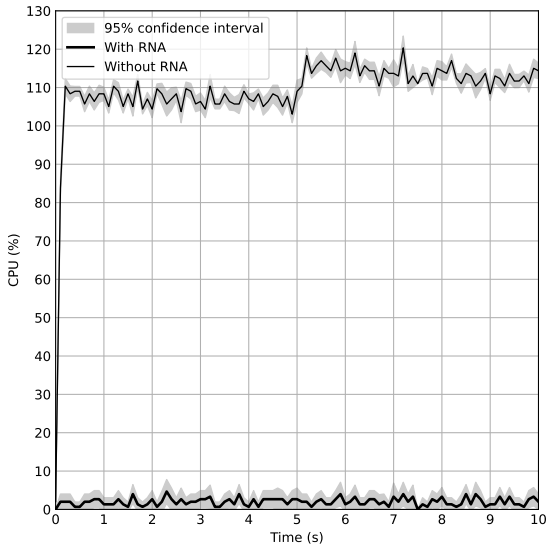
[2]Usage of one CPU core. 100% represents usage of a full core, 150% represents usage of one and a half CPU cores.

In Figure 5.5, at time $5\,\mathrm{s}$, we observe a significant change in the increase of memory and CPU usage. Our hypothesis is that this is the moment internal Zeek buffers are filled, decreasing the rate packets are read, slowing down the increase of memory usage, and increasing the CPU usage. In a real-world scenario, network operators would not allow an IDS to drop packets and would increase the processing power or implement a sampling strategy. Our intention with this comparison is to contrast the usage of a normal Zeek deployment compared to our RNA framework, which uses Programmable Data Planes to offload IDS operations. Nevertheless, our results suggest that the observed reduction in CPU usage for the RNA-based setup would allow for the use of a smaller cluster for the given scenario.

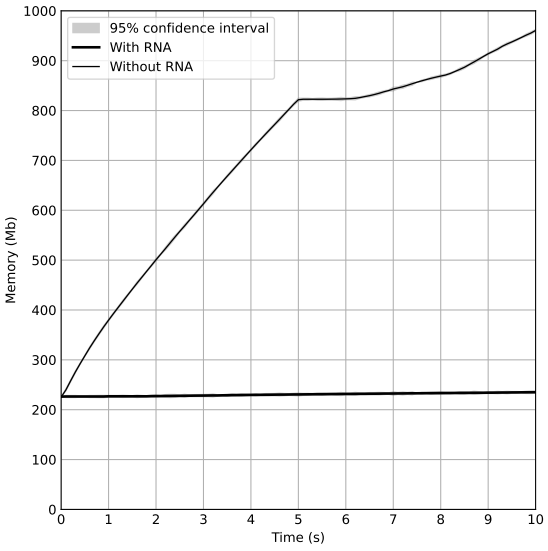
Another important note is that the Dynamic Protocol Detection (DPD), which we presented in Figure 2.2, is unable to dynamically detect protocols when RNA is used, potentially resulting in better performance, giving RNA an advantage. This is also one of the assumptions we made to simplify the development of the mechanism.

Figure 5.5 – RNA Performance Evaluation

(a) CPU usage by time



(b) Memory usage by time



Source: the author (2022).

6 CONCLUSION

In this project, we investigated the benefits of using Programmable Data Planes to offload Zeek monitoring scripts. We also took the first step towards an automatic code generation mechanism, which enables any network operator without programming knowledge to offload Zeek scripts to programmable forwarding devices. We implemented an automatic code generator that identifies which Zeek Events are required by a set of scripts and, using templates, automatically generates P4 and Zeek code to offload these scripts.

After proposing additions to the RNA framework and implementing our prototypical automatic code generator, we evaluated the proposed approach and assessed its capabilities of automatically generating code and enhancing performance. We showed the mechanism generates almost 3 thousand lines of code, which, otherwise, a developer would need to write manually in order to offload four Zeek Scripts. We demonstrated that RNA can give a performance benefit compared to server-based intrusion detection, resulting in $57\times$ less CPU usage and $4\times$ less memory usage for the workload used in the experiments. Moreover, we have also shown that our approach can produce these benefits for network operators without any previous P4 programming knowledge. It is also important to note that these results are still to be confirmed with future experiments using hardware PFDs.

This project took the first step towards adding automatic code generation to RNA. Next, we describe some opportunities for future work and items that could be improved. These items are both suggestions for RNA and the code generator mechanism.

Stateful Connections

The next step to allow better protocol handling, mainly of TCP connections, is the implementation of stateful analysis into RNA. Stateful analysis will allow far more significant performance benefits if executed in the data plane. It will offload a significant part of Zeek state management, allowing P4 to analyze more connection-based protocols.

Multiple Offloaders

At the current state of RNA and our code generation mechanism, it is not possible to trigger more than one Offloader per incoming packet. This limitation could impact future deployments where many scripts are being executed. This is a problem that should

be solved in the future.

Enhanced Zeek Connection Management

Zeek uses an object called *Connection* (not to be confused with an actual *network connection*) to manage sessions internally. This object is not deallocated after usage in our approach. Future work must develop a mechanism to handle protocol timeouts and free resources after these are no longer needed. Some protocols also have timeout events, which should also be triggered by the PDP.

Security analysis

While evaluating RNA and the code generation mechanism, we did not consider a scenario where an attacker would try to attack Zeek through RNA or RNA itself. In future projects, a security assessment should be performed to ensure that using RNA does not introduce vulnerabilities to the infrastructure that attackers could exploit.

REFERENCES

ADAMS, D. **Brute Force Against SSH and FTP Services**. 2019. Available from Internet: <https://linuxhint.com/bruteforce_ssh_ftp/>. Accessed in: 2022-08-23.

Akamai Technologies. **Akamai Ransomware Threat Report**. 2022. Available from Internet: <https://www.akamai.com/resources/research-paper/akamai-ransomware-threat-report>. Accessed in: 2022-08-19.

BOSSHART, P. et al. P4: Programming Protocol-independent Packet Processors. **ACM SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833.

CAIDA. **The CAIDA UCSD Anonymized Internet Traces 2016**. 2016. Available from Internet: <https://catalog.caida.org/dataset/passive_2016_pcap>. Accessed in: 2022-08-16.

Cisco Systems. **Snort - Network Intrusion Detection and Prevention System**. 2022. Available from Internet: <https://www.snort.org/>. Accessed in: 2022-08-16.

Cloudflare, Inc. **NTP amplification DDoS attack**. 2022. Available from Internet: <https://www.cloudflare.com/learning/ddos/ntp-amplification-ddos-attack/>. Accessed in: 2022-08-23.

CORDEIRO, W. L. da C.; MARQUES, J. A.; GASPARY, L. P. Data Plane Programmability Beyond OpenFlow: Opportunities and Challenges for Network and Service Operations and Management. **Journal of Network and Systems Management**, v. 25, n. 4, p. 784–818, oct. 2017. ISSN 1573-7705.

DREGER, H. et al. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In: KEROMYTIS, A. D. (Ed.). **15th USENIX Security Symposium (USENIX Security 06)**. Vancouver, BC, Canada: USENIX Association, 2006. p. 257–272. Available from Internet: <https://www.usenix.org/conference/15th-usenix-security-symposium/dynamic-application-layer-protocol-analysis-network>.

GUPTA, R. **Traceroute in Network Layer**. 2021. Available from Internet: <https://www.geeksforgeeks.org/traceroute-in-network-layer/>. Accessed in: 2022-08-16.

HJSON, a user interface for JSON. 2022. Available from Internet: <https://hjson.github.io/>. Accessed in: 2022-08-05.

HUMMEL, R.; HILDEBRAND, C. **NETSCOUT Threat Intelligence Report - Issue 7: Findings From 1H 2021 - The Long Tail of Attacker Innovation**. Westford, MA, USA, 2021. Retrieved on 2022-02-23. Available from Internet: <https://www.netscout.com/threatreport>. Accessed in: 2022-08-19.

ILHA, A. da S. **Towards a General Approach for Cyberattack Detection Using Programmable Data Planes**. Dissertation (Master) — Universidade Federal do Rio Grande do Sul, 2022.

JACOBSON, V.; LERES, C.; MCCANNE, S. **libpcap**. 1994. Lawrence Berkeley Laboratory, Berkeley, CA. Available from Internet: <https://www.tcpdump.org/>. Accessed in: 2022-08-16.

KIM, C.; LEE, J. **Programming the network dataplane. ACM SIGCOMM Tutorial.** 2016. Available from Internet: <https://conferences.sigcomm.org/sigcomm/2016/files/program/netpl/netpl16-kim.pdf>.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In: XIE, G. G. et al. (Ed.). **Proceedings of the 9th ACM Workshop on Hot Topics in Networks. HotNets 2010, Monterey, CA, USA - October 20 - 21, 2010**. ACM, 2010. p. 19. Available from Internet: <https://doi.org/10.1145/1868447.1868466>.

NTP Monlist Detection. 2021. Available from Internet: <https://github.com/dopheide-esnet/zeek-ntp-monlist/>. Accessed in: 2022-08-14.

PAXSON, V. Bro: a system for detecting network intruders in real-time. **Computer Networks**, v. 31, n. 23, p. 2435–2463, 1999. ISSN 1389-1286. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S1389128699001127>.

REARDON, B. **Pingback C2 Detection**. 2021. Available from Internet: <https://github.com/corelight/pingback/>. Accessed in: 2022-08-14.

Science Direct. **Network Based Intrusion Detection System**. 2022. Available from Internet: <https://www.sciencedirect.com/topics/computer-science/network-based-intrusion-detection-system>. Accessed in: 2022-08-23.

SONG, H. Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane. In: **Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: Association for Computing Machinery, 2013. (HotSDN '13), p. 127–132. ISBN 9781450321785. Available from Internet: <https://doi.org/10.1145/2491185.2491190>.

TAVARES, P. **Pingback malware: How it works and how to prevent it**. 2021. Available from Internet: <https://resources.infosecinstitute.com/topic/pingback-malware-how-it-works-and-how-to-prevent-it/>. Accessed in: 2022-08-23.

The Open Information Security Foundation. **Home - Suricata**. 2022. Available from Internet: <https://suricata.io/>. Accessed in: 2022-08-16.

The P4 Language Consortium. **p4app**. 2019. Available from Internet: <https://github.com/p4lang/p4app>.

The P4 Language Consortium. **BMv2**. 2020. Available from Internet: <https://github.com/p4lang/behavioral-model>.

The Zeek Project. **Signature Framework – Book of Zeek**. 2022. Available from Internet: <https://docs.zeek.org/en/master/frameworks/signatures.html>. Accessed in: 2022-08-23.

The Zeek Project. **The Zeek Network Security Monitor**. 2022. Available from Internet: <https://zeek.org/>. Accessed in: 2022-08-16.

The Zeek Project. **The Zeek Network Security Monitor**. 2022. Retrieved on 2022-01-03. Available from Internet: <https://docs.zeek.org/en/master/about.html>.