



Assignment of master's thesis

Title:	Network Traffic Analysis Using Weak Indicators
Student:	Bc. Richard Plný
Supervisor:	Ing. Karel Hynek, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

Study the most common strategies of network traffic classification and analysis. Study a weak-indication approach for network classification used in [1] and propose several weak indicators usable in network traffic analysis. Create a software library simplifying the development of network classifiers based on weak indicators. Create a prototype of a network classifier using the developed library and NEMEA system [2,3]. Evaluate the prototype based on standard performance measures—classification accuracy, speed, and memory requirements.

[1] D, Uhříček, K. Hynek, T. Čejka, D. Kolář. "BOTA: Explainable IoT malware detection in large networks". IEEE Internet of Things Journal. (2022)

[2] T. Čejka, V. Bartoš, M. Svejpes, Z. Rosa, and H. Kubatova, "NEMEA: A Framework for Network Traffic Analysis," in 12th International Conference on Network and Service Management (CNSM 2016), Montreal, Canada, 2016.

[3] <https://github.com/CESNET/NEMEA>



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Master's thesis

Network Traffic Analysis Using Weak Indicators

Bc. Richard Plný

Department of Information Security
Supervisor at CTU: Ing. Karel Hynek, Ph.D.
Supervisor at TalTech: Karl-Erik Karu MSc.

May 9, 2024

Acknowledgements

I am deeply thankful to Karel Hynek for giving me keys to the office, Karl-Erik Karu for letting me win in Billiards and his girlfriend Sibel for coffee, George Orwell, some random people I met during my Erasmus+ studies, Pavel Šiška for doing the code reviews in memes, Princess Diana, Tomáš Čejka for his tea recommendations, and my colleagues for their professional toleration of my drawings on the whiteboard. I would also like to express my biggest thanks to my parents for not kicking me out of the house, Airbus engineers, my sister for constantly annoying me, Jaromír Jágr, my friends for always eating my chips, Kaja Kallas, the prime minister of Estonia, Martin Růžička for buying a grill, the Vietnamese restaurant across the street, Bohumil Říha, the author of the book *Honzíkova cesta*, and Mother Theresa. My thanks also go to Mr. Ryan Gosling for his acting in *Drive* (2011), the lady who sold me a rain jacket, Felix Holtzmann and his wife Eva, and the emperor Josef I. for establishing the Czech Technical University in Prague. Last but not least, I would like to acknowledge everyone who supported me during the years, God, and, of course, John Cleese for giving me the idea with his "thanks everyone on the planet" speech.

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 Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 9, 2024

Czech Technical University in Prague

Faculty of Information Technology

© 2024 Richard Plný. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Plný, Richard. *Network Traffic Analysis Using Weak Indicators*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstrakt

Tato práce popisuje metody pro klasifikaci síťového provozu, detekci hrozeb a bezpečnostního monitorování počítačových sítí. Poté je představena nová knihovna s názvem Weak Indication Framework, která si klade za cíl usnadňovat vývoj nových detektorů. Knihovna také umožňuje vyvíjet výkonné detekční systémy pro vysokorychlostní sítě. Dále je vyvinuto několik detektorů s pomocí této knihovny jako demonstrace její použitelnosti. Všechny detektory byly navíc úspěšně nasazené na národní síť CESNET3 s více než půl milionem uživatelů.

Klíčová slova klasifikace síťového provozu, detekce hrozeb, bezpečnostní monitorování sítě, slabé indikátory

Abstract

This thesis studies methods for network traffic classification, threat detection, and network security monitoring. Moreover, a new software library called Weak Indication Framework (WIF) is introduced to ease the development of new threat detector systems. The WIF supports systems with high explainability and high performance for high-speed networks. Several detectors are also developed to demonstrate the usability of the WIF. Furthermore, all developed detectors were successfully deployed to the national network CESNET3 with more than half a million users.

Keywords network traffic classification, threat detection, network security monitoring, weak indications

Contents

Introduction	1
1 Network Communication Protocols	3
1.1 ISO/OSI Model	3
1.2 TCP/IP Model	4
1.3 Description of Selected Protocols	6
1.3.1 IP	6
1.3.2 TCP	6
1.3.3 UDP	9
1.3.4 TLS	9
1.3.5 DNS	10
1.3.6 OpenVPN	10
1.3.7 WireGuard	11
2 Network Monitoring	13
2.1 Deep Packet Inspection	13
2.2 Flow-based Monitoring	14
2.2.1 Flow Exporter	14
2.2.2 Flow Collector	15
2.3 Network Traffic Analysis	16
2.3.1 Threat Detection	16
2.3.2 NEMEA	17
3 Classification Methods	19
3.1 Pattern Matching	19
3.2 Machine Learning	21
3.2.1 Machine Learning in Network Domain	22
3.2.2 Description of Selected Supervised ML Models	22
3.2.3 Description of Selected Unsupervised ML Models	23
3.3 Anomaly Detection	24
3.4 Data Fusion	25
3.4.1 Kalman Filter	26
3.4.2 Dempster-Shafer Theory	27
3.5 BOTA	27
3.6 DeCrypto	28

3.7	VPN Detection	28
4	Weak Indication Framework	31
4.1	Introduction	31
4.2	Development Environment & Process	33
4.3	Architecture	34
4.4	Data Structures	36
4.5	Classifiers	38
4.5.1	IP Prefix Classifier	39
4.5.2	Regex Classifier	40
4.5.3	Scikit ML Classifier	41
4.5.4	ALF Classifier	42
4.6	Combinators	42
4.6.1	Basic Combinators	43
4.6.2	Binary DST Combinator	43
4.7	Reporters	43
4.7.1	Unirec Reporter	44
4.8	Utils	44
4.9	Discussion	45
5	Network Classification Prototypes	47
5.1	Introduction	47
5.2	Cryptomining Detector	47
5.3	Tor Detector	49
5.4	Tunnel Detector	50
5.4.1	Data Stores	51
5.4.2	Weak Detectors	51
5.4.3	Export	52
5.4.4	Build Dependencies	53
5.5	Malware Detector	54
5.5.1	Data Stores	54
5.5.2	Weak Detectors	55
5.5.3	Export	56
5.5.4	Build Dependencies	57
5.6	Discussion	57
6	Evaluation	59
6.1	Metrics & Methodology	59
6.2	Cryptomining Detector	60
6.2.1	Correctness	60
6.2.2	Throughput	60
6.3	TorDer	61
6.3.1	Correctness	61
6.3.2	Thoughtput	62
6.4	TunDer	62
6.4.1	Correctness	62
6.4.2	Throughput	63
6.5	MalDer	64
6.5.1	Correctness	64
6.5.2	Throughput	64

6.6	Discussion	64
7	Deployment	67
7.1	Targeted Network	67
7.2	Cryptomining Detector	68
7.3	Tor Detector	68
7.4	Tunnel Detector	69
7.5	Malware Detector	69
7.6	Summary	69
	Conclusion	71
	Bibliography	73
A	Evaluation	79
A.1	Comparison of DeCrypto implementations	79
B	Deployment Notes	81
B.1	Alerts from Cryptomining Detector	81
C	User Manual	83
C.1	Cryptomining Detector	83
C.2	Tor Detector	85
C.3	Tunnel Detector	86
C.4	Malware Detector	88
D	Contents of the Attached Archive	89

List of Figures

1.1	ISO/OSI model	3
1.2	TCP/IP and ISO/OSI model comparison	5
1.3	IPv4 Packet	7
1.4	IPv6 Packet	7
1.5	TCP Header	8
1.6	TCP Connection Management	8
1.7	TCP Re-transmit	9
1.8	UDP Header	9
1.9	TLS handshake	10
1.10	OpenVPN tunnel	11
2.1	Flow-based network monitoring	14
2.2	IPFIX messages	15
2.3	Threat detection pipeline	17
2.4	NEMEA System	17
3.1	ML Model Development Process	21
3.2	Finite automaton for OpenVPN detection	29
4.1	High-level view of WIF object groups	32
4.2	Example of WIF-based detection module	35
4.3	Intended use of the Weak Indication Framework	36
4.4	WIF-ALF interconnection	42
4.5	Process of <code>TimerCallback</code> utilization in <code>Timer</code> class	45
5.1	Architecture of the Cryptomining detector	48
5.2	Architecture of the Tor detector	49
5.3	TunDer architecture	51
5.4	MalDer architecture	54
7.1	National network CESNET3	67
7.2	Screenshot of alert in the Mentat web interface	68
7.3	Server with deployed detectors	69
B.1	Number of alerts from the deployed Cryptomining detector	81

List of Tables

- 1.1 TCP/IP Overview 5

- 4.1 Supported data types by DataVariant class 36
- 4.2 Performance of DeCrypto systems 41

- 6.1 Parameters of the machine used for testing 60
- 6.2 DeCrypto parameters used for evaluation 60
- 6.3 DeCrypto results 61
- 6.4 Performance of DeCrypto systems 61
- 6.5 Performance of Tor detector 62
- 6.6 TunDer rule tests 63
- 6.7 Performance of TunDer detector 63
- 6.8 MalDer rule tests 64
- 6.9 Performance of MalDer detector 65

- A.1 Comparison of DeCrypto implementations on the Design dataset . . 79
- A.2 Comparison of DeCrypto implementations on the Evaluation dataset 80

List of Listings

1	Interface of the <code>FlowFeatures</code> class	37
2	Interface of <code>IpAddress</code> class	38
3	Common interface of the <code>Classifier</code> objects	39
4	Bridge module used by Scikit ML Classifier	41
5	Common interface of the <code>Combinator</code> objects	43
6	Common interface of the <code>Reporter</code> objects	44
7	<code>TimerCallback</code> Interface	45
8	<code>Rule</code> Interface	53

Introduction

Most Internet traffic is encrypted nowadays, meaning knowing and understanding what is happening in our computer networks is non-trivial. However, we must still protect our users and infrastructure, which is only possible with network monitoring. Moreover, network monitoring is essential for discovering security threats, outgoing attacks, and more. Many detection and network traffic classification methods were developed for this purpose. Unfortunately, network traffic evolves: new communication protocols (for example, QUIC and HTTP/3) are being developed, traffic of newly-developed applications can cause an existing detector to produce false outputs, and even new versions of existing applications can change the way the application communicates over a computer network, and an existing detector may stop working. Therefore, new detection and classification methods must be developed to respond to the changing environment of computer networks.

One of the goals of this thesis is to perform a thorough review of current classification and detection methods and to identify the most used principles in the field. The main goal of this thesis is to design and implement a library, Weak Indication Framework (WIF), that contains the identified commonly used methods, making the development of new network threats and application detectors much easier and faster. Moreover, it will reduce the reaction time between discovering a new traffic type on a computer network and deploying a detector for the newly discovered traffic. Other goals include developing several Proof-of-Concept detectors based on the newly designed WIF and thorough testing both functionality and performance.

This thesis was developed in coordination with CESNET, the Czech National and Research Network (NREN) operator. CESNET operates the national research and educational network CESNET3, which has 100+ Gbps lines and half a million daily users. WIF and WIF-based detectors primarily target the CESNET3 network in terms of performance, the information contained in the output alerts, and other use cases. WIF-based software will target national-level high-speed networks and provide the ability to process thousands of network flows per second.

Network Communication Protocols

Understanding network traffic and communication protocols is essential before attempting to design detection methods. Therefore, this chapter provides basic principles of communication over computer networks together with a brief description of chosen communication protocols. Protocols were chosen with emphasis on the thesis topic and detection methods described later.

1.1 ISO/OSI Model

ISO/OSI model is an abstract description of interprocess communication [1]. Day and Zimmermann [1] further explain that this model divides communication into 7 abstract layers, which are shown on the Figure 1.1. The layer names (from the top) are Application, Presentation, Session, Transport, Network, Data Link, and Physical. As also stated in [1], each layer is independent and provides services to higher layers. Therefore, the operation of each layer can be changed, provided that the offered service will remain the same [1].

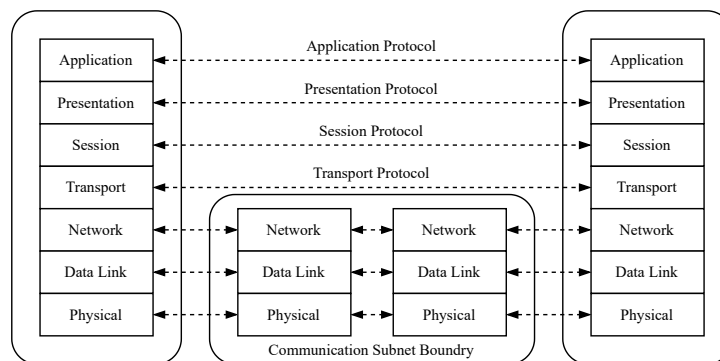


Figure 1.1: ISO/OSI model

Application layer is the most top layer, and therefore, it does not provide services to higher layers [1]. It is the only one communicating directly with the end user and defines services presented at the user-end [2]. The end user typically interacts directly with software (web browsers, email clients, ...), which initiates communication.

Presentation layer transforms data into a specific format and encapsulates it into messages, which are then passed to the next layer. As explained in [2], it is responsible for the way data is presented to the application. Moreover, encryption, translation, and compression are realized by the presentation layer [2].

Session layer manages connections (a link or a path between two computers is called a connection). It has the ability to organize multiple connections at once [1, 2]. The connection between two computers has to be created, controlled, kept alive, and destroyed at the end. It is also possible that this layer performs user authentication and token management [2].

Transport layer provides protocols and procedures for data transfer from one host to another [1]. Protocols of this layer are divided into connection-oriented and connectionless [1, 2]. Data is divided into smaller chunks called *segments*, which is the Processing Data Unit (PDU) of this layer [2]. Segment must be small enough, so network-layer and transport-layer headers can be added. The transport layer provides end-to-end error detection and recovery, sequence control, segmentation, and more [2].

Network layer provides independent data transfer technology, such as relaying and routing decisions [1]. PDU of this layer is called *packet*. The network layer provides protocols for packet transfer between nodes and different networks. Every node (a computer connected to the network) has an address, which can be used as a unique identifier [2]. One node can send data to another by providing the destination address. The network will find a way to deliver the packet to the correct destination — by routing process.

Data Link layer provides data transfer between two hosts on the *same* computer network [1]. It is able to detect and potentially correct errors that could occur on the physical layer [1, 2]. PDU used on the Data Link layer is called *frame*. It also provides identification and parameter exchange possibilities, which are needed for communication to take place [2].

Physical layer performs the actual transmission (transmit and receive) of raw data between devices. As mentioned in [1], it provides electrical, mechanical, and other functionalities to access physical mediums.

1.2 TCP/IP Model

TCP/IP model, also known as the Internet protocol suite, is a framework of organized communication protocols used in the Internet [3, 4]. TCP/IP is a practical model which relies on standardized protocols. On the other hand, ISO/OSI serves as a comprehensive protocol-independent framework and is used only in theory. TCP/IP is also a layered model [4] and very similar to the ISO/OSI model. Comparison is shown on the Figure 1.2.

In the TCP/IP framework, the Application, Presentation and Session layers of ISO/OSI are grouped together and they form one layer called the Application layer. Moreover, Data Link and Physical are grouped into the Link layer. The layer names (from the top) in the TCP/IP model are: Application, Transport, Network, and Link layer [5].

Application layer is the most upper layer and provides protocols used directly by most software. It includes HTTP (web surfing), FTP (file transfer),

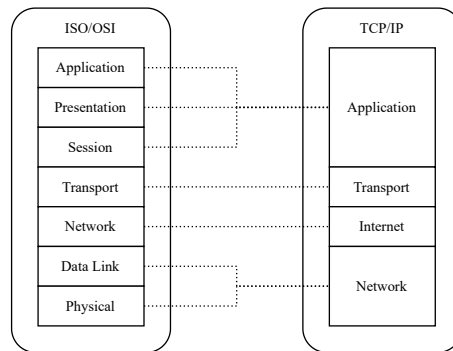


Figure 1.2: TCP/IP and ISO/OSI model comparison

Table 1.1: TCP/IP Overview

No.	Layer	Protocols	Data Unit
4	Application	HTTP, FTP, SMTP	APDU
3	Transport	TCP, UDP, QUIC	Stream, Datagram, Segment
2	Internet	IP	Packet
1	Link	Ethernet	Frame

SMTP (emails), and many more protocols [2]. It handles data representation and encoding, and it also interacts with end-user [2].

Transport layer contains connection-oriented (TCP) and connectionless (UDP) protocols [2]. Protocols of this layer can communicate with other hosts independently on which protocol is used on the lower layer. Segmentation and error control (prevention of network overload) are provided on this layer. TCP provides reliability over non-reliable IP protocol [2]. However, reliability is not needed in some cases. UDP provides a much smaller communication overhead and is suitable for real-time applications [2].

Internet layer performs unreliable data transmission from a source host to a destination host (even between different networks) [4]. This is achieved by the routing process, host addressing and IP addresses [4]. The most used protocol of this layer is IP protocol. Two addressing systems defined for network host identification are: IPv4 (32-bit IP address) and IPv6 (128-bit IP address) [6].

Link layer performs transmission (send and receive) on the local network — the network, to which a computer is connected to. It contains protocols and methods used by a host to access a network [4]. Media Access Control (MAC) addresses are used for host identification on the local network [2]. Moreover, data is divided into frames, which are then transmitted by a network interface card (NIC) [2].

A summary of the TCP/IP framework and its protocols is available in Table 1.1.

1.3 Description of Selected Protocols

This section contains a brief description of selected communication protocols. The selection was performed based on relevancy to the thesis topic.

1.3.1 IP

Internet Protocol (IP) is a Network layer protocol initially defined by RFC 791 [7]. It is used for the transmission of packets over IP networks with the use of a routing process. Internet Protocol version 4 (IPv4) is the most used protocol on the Internet, developed in 1978. Internet Protocol version 6 (IPv6) [8] is the newer version from 1996. However, IPv6 is being adopted on the Internet very slowly, and IPv4 remains dominant.

Hosts on IP networks are labeled by numerical IP addresses, which are then used for “navigation” in IP networks. IPv4 uses 32-bit addresses, usually written in decimal system, for example 192.168.0.1. IPv4 can provide almost 4.3 billion different addresses. IPv6 uses 128-bit addresses, usually written in hexadecimal system, for example 2001:0db8:85a3:0000:0000:8a2e:0370:7334.

Moreover, both IPv4 and IPv6 can be divided into subnets. IP subnets are defined by network address and number of bits used for network prefix, usually written in CIDR format: 10.0.0.0/24 defines an IP subnet, in which all IP addresses have prefix 10.0.0. The last 8 bits are used for local identification. The 10.0.0.0/24 defines a subnet with IP addresses ranging from 10.0.0.0 to 10.0.0.255. Subnets can also be specified by network address and subnet mask. The same subnet can be defined as 10.0.0.0 and its subnet mask 255.255.255.0.

The initial idea was that each device connected to the Internet would have its own unique (public) IP address. However, it was assessed that not every device needs a public IP address, for example, for security reasons. Therefore, some IP subnets were declared as private. Network devices do not perform routing for these addresses and they rather use Network Address Translation (NAT) before transmission to the next IP network.

IP protocol also defines a structure that holds data during transmission, called an IP packet. Each packet has two components: a header and a payload. The IP header of version 4, version 6 is shown on the Figure 1.3, Figure 1.4. Packet payload contains application data. Usually, it is a datagram of a protocol from a higher layer, for example, a TCP segment or UDP datagram. It is a connectionless protocol, potentially unreliable. Reliability is typically ensured by a higher-layer protocol, such as TCP.

1.3.2 TCP

Transmission Control Protocol (TCP) is a protocol working on the Transport layer defined by RFC 793 [9]. It provides a reliable connection between two hosts over an IP network for delivering streams of bytes. TCP was developed in 1974 by Vint Cerf and Bob Kahn [10].

TCP accepts a data stream, divides it into chunks, and prepends the TCP header to each one, creating TCP segments. TCP segment is then passed to the IP protocol and sent. The structure of the TCP header is shown on the Fig-

1.3. Description of Selected Protocols

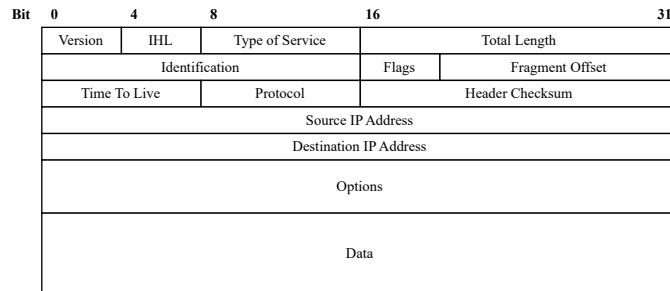


Figure 1.3: IPv4 Packet

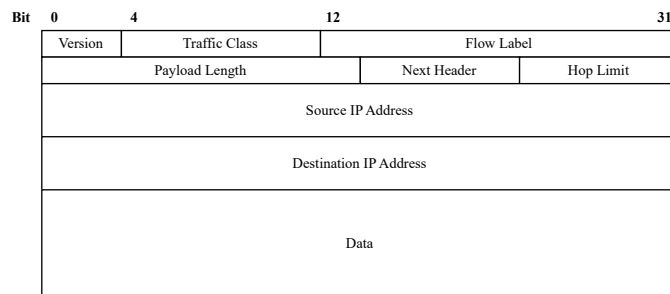


Figure 1.4: IPv6 Packet

Figure 1.5. TCP flags indicate a particular state and hold additional information about the connection. TCP flags are explained below [11]:

CWR Congestion window reduced: Used to indicate that a TCP segment with ECE flag set was received.

ECE ECN-Echo: used together with SYN flag. A TCP segment with both ECN and SYN flags set means that the host supports ECN. If the SYN flag is not set, it indicates that the network is congested.

URG Urgent: TCP segment holds urgent data which should be processed as soon as possible.

ACK Acknowledgement: sent by the receiver to the sender to confirm that data was received successfully.

PSH Push: indicates that received buffered data should be pushed to the receiving application.

RST Reset: used to reset the connection. The host informs the other one that it will not receive or send any more data.

SYN Synchronization: used for synchronization of sequence numbers for packet labeling. Usually, SYN should be only sent once and as the first packet by each side.

FIN Finalize: used to close the connection. TCP segment with FIN set is the last packet from the sender.

Source Port		Destination Port	
Sequence Number			
Acknowledgement Number			
DO	RSV	Flags	Window
Checksum		Urgent Pointer	
Options			

Figure 1.5: TCP Header

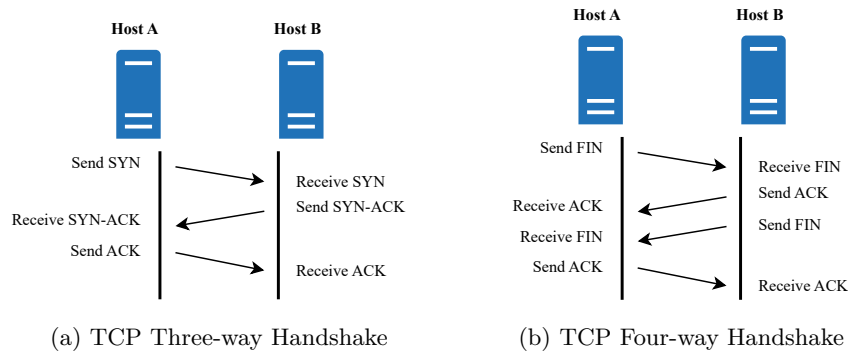


Figure 1.6: TCP Connection Management

A three-way handshake is used to establish a reliable connection. Firstly, host A sends a TCP message with the SYN flag set. Host B replies with a TCP message with flags SYN and ACK set. Host A replies with a TCP message with an ACK flag. The process is shown on Figure 1.6a.

Termination of connection is performed by a four-way handshake. Firstly, host A sends a TCP message with the FIN flag. Host B replies with a TCP message with an ACK flag. Then, host B sends another TCP message with the FIN flag. After host A replies with a TCP message with an ACK flag, the connection is terminated on both sides. See Figure 1.6b with visualized four-way handshake.

IP packets can be lost during transmission due to network overload, errors, and more. Moreover, packets can be duplicated and arrive in incorrect order. TCP can detect these problems and handle them. TCP segment header contains the Sequence number, which is used for the detection of missing data. The receiving side is sending acknowledgments (ACKs) to confirm the data was received. If the receiving side does not send ACK, the sending side will transmit the same data again, as shown on the Figure 1.7. Furthermore, Sequence number defines an unambiguous order of data. Therefore, the receiving side is assured to have all data in the correct order.

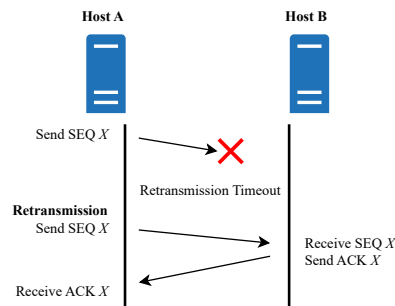


Figure 1.7: TCP Re-transmit

1.3.3 UDP

User Datagram Protocol (UDP) is an alternative to TCP protocol on the Transport layer. It was designed by David P. Reed in 1980 and defined in RFC 768 [12]. UDP does not require a three-way handshake or any other prior communication compared to the TCP. UDP is a simple, connectionless protocol. It does not contain ordering or deduplication checks. Moreover, there is no guarantee of delivery.

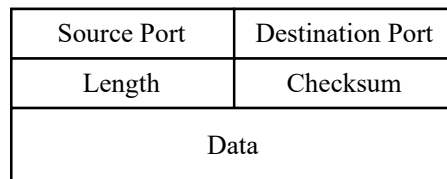


Figure 1.8: UDP Header

UDP header, shown on the Figure 1.8, only contains several fields. It contains source and destination ports for application identification. A checksum is used for error-checking. Since it adds a minimal overhead, it is typically used in real-time applications, such as voice and video streaming.

1.3.4 TLS

Transport Layer Security (TLS) is a cryptographic protocol operating above the Transport layer. It was initially introduced by RFC 2246 [13] and its current version is TLS 1.3 defined by RFC 8446 [14]. Cryptography means providing confidentiality (privacy), integrity and authenticity. It has two layers: TLS handshake and TLS record protocols. TLS-secured communication cannot be eavesdropped and tampered with. During the connection establishment, asymmetric cryptography is used to establish shared key and cryptographic algorithm parameters, which are then used for faster symmetric cryptography and data exchange.

TLS connection starts with a TLS handshake, during which a server is authenticated by its certificate. Moreover, protocol versions and lists of supported ciphers by both sides are exchanged. The TLS version and the cipher supported by both sides are chosen. TLS handshake is shown on the Figure 1.9.

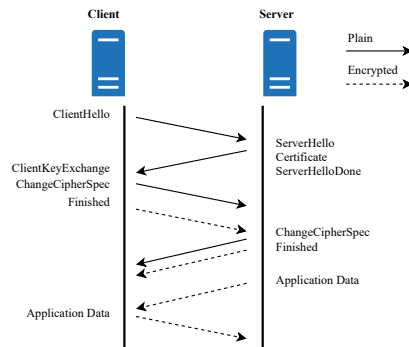


Figure 1.9: TLS handshake

TLS also offers lots of extensions, such as Server Name Indication (SNI). A client uses SNI to indicate, to which server it is trying to connect. It is sent during the TLS handshake. TLS SNI is helpful when multiple (web) servers are running on the same IP address and port. The server can present a correct certificate for a hostname that is being accessed.

1.3.5 DNS

Domain Name System is a hierarchical naming system of computers, introduced by RFCs 1034 [15] and 1035 [16]. Domain Name System (DNS) protocol is used for resolving domain names to IP addresses of corresponding servers. Each domain has its own authoritative name servers. Management of sub-domains can be delegated to other name servers. For example, when a user is accessing `www.google.com`, the authoritative name server for `com` is queried first. The query is delegated to the name server for `google`, which provides an IP address, where the user should connect to access the `www.google.com` service. DNS typically uses UDP protocol.

1.3.6 OpenVPN

OpenVPN [17] is a tunneling protocol used in virtual private networks (VPNs). It provides authentication of peers and also confidentiality by using encryption. It is used to securely access private corporate networks from home and other remote resources. Moreover, OpenVPN can be used to access the Internet privately over an unsecured network.

Firstly, OpenVPN establishes a secured connection (tunnel) to a VPN server. Regular traffic is then transmitted through this tunnel. Packets are passed to the OpenVPN protocol; each packet receives a new OpenVPN protocol header and is sent through the tunnel. The destination VPN client unwraps such packets and retrieves the original ones. Then, each packet is processed and routed as it would have been in the regular send process. The process is shown on the Figure 1.10.

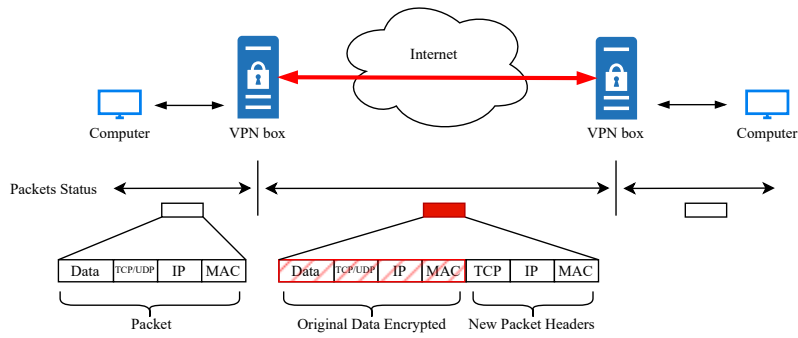


Figure 1.10: OpenVPN tunnel

1.3.7 WireGuard

WireGuard [18] is another protocol for VPN tunneling. It uses UDP as an underlying protocol to provide better performance and higher speed than OpenVPN. It works similarly to OpenVPN and other tunneling protocols. WireGuard also uses state-of-the-art cryptography, such as Curve25519 and ChaCha20. Moreover, it can be extended by third-party scripts.

Network Monitoring

Once we understand how computers talk to each other and what is transmitted over computer networks, it is essential to understand how computer networks can be monitored. This chapter provides a brief introduction to network monitoring approaches, and their pros and cons, with an emphasis on high-speed network monitoring. Furthermore, threat detection and response principles are briefly discussed.

2.1 Deep Packet Inspection

Deep Packet Inspection (DPI) is a technique for data processing which inspects data carried by the packet [19]. It tries to understand the structure and meaning of the data. DPI is used for exploring the behavior of network applications, network performance troubleshooting, malformed packet detection, threat detection, and more [19].

As explained by El-Maghraby et al. [19], DPI is looking not only at protocol headers but also at the packet payload. It can recognize traffic types based on the headers (TCP, UDP, ...) and identify the communicating sides (based on the IP header). Moreover, the Application layer header can be recognized as well. Therefore, DPI can mark traffic as HTTP, SMTP, DNS and other types. DPI can also verify that headers contain valid information and that the packet is well-formed. Detection of an ill-formed packet can mean that someone is trying to break into the system by exploiting a known vulnerability — for example, the TLS Heartbleed exploit. DPI is able to detect such attempts.

The packet payload is also inspected [19]. DPI can detect data exfiltration, botnet communication, and more. Similar to ill-formed packet headers, DPI can detect ill-formed SQL queries and possible attempts to perform SQL injection. Furthermore, it can be used to restrict access to certain websites in corporate networks, such as social media, adult-content sites, and more [20].

DPI is a very useful tool for network monitoring. It can classify network traffic and discover possible threats and attacks [19]. However, there are several drawbacks as well. Firstly, DPI requires network traffic to be unencrypted — meaning that encryption is not used at all, or traffic is decrypted by the system performing DPI. This raises huge concerns regarding user privacy and security [20]. Moreover, DPI can be used for censorship by oppressive regimes.

DPI also needs to have full packets available for analysis which creates another problem: scalability. It is feasible to use DPI in home and small business networks because network traffic load stays in reasonable numbers. However, corporate and ISP-level networks with 100+ Gbps lines produce huge amounts of traffic (packets) that must be processed. Moreover, processing time needs to be low; otherwise, users would experience long round trip times [21]. Therefore, operating DPI on large networks is demanding and might be impossible [22].

2.2 Flow-based Monitoring

Flow-based network monitoring is a prevalent method for monitoring of high-speed networks [22]. As explained by Čeleda et al. [22], it focuses on the analysis of flows, rather than packets. The basic idea is that rather than processing full packets, packets are aggregated into “flows” based on common characteristics. A flow is defined as “a set of packets or frames passing an Observation Point in the network during a specific time interval” [23]. The amount of data is significantly lowered and monitoring of large networks is possible [22]. However, since packets are aggregated into flows, less information is available for analysis than in the DPI approach. A typical flow-based network monitoring setup is shown in the Figure 2.1.

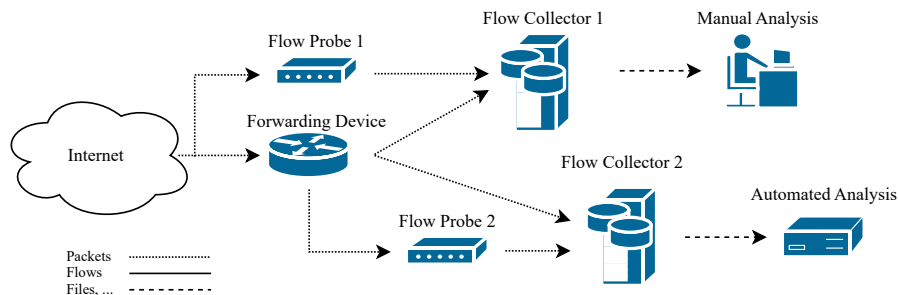


Figure 2.1: Flow-based network monitoring

2.2.1 Flow Exporter

Firstly, traffic copy passing through the monitored line is created. This traffic is sent to the server where a so-called flow exporter is running. A flow exporter (flow probe) is a software where Flow Metering and Exporting processes take place [24]. During the Metering process, packets are aggregated into flows, and exported during the Exporting process. Therefore, flow exporters and probes create IP flows.

$$\text{FlowKey} = \text{Hash}(\text{SRC_IP}, \text{DST_IP}, \text{SRC_PORT}, \text{DST_PORT}, \text{PROTO}) \quad (2.1)$$

Packets are aggregated by using a flow key, which is usually a hash of the 5-tuple: source and destination IP addresses, source and destination ports, and the protocol used at the Transport layer [22, 24, 25], also shown on equation 2.1. In this case, a flow will contain information about communication between two

communicating applications. However, the flow key and aggregation procedure can be defined in various ways. Fioreze et al. [25] proposed multiple sets of properties which produce IP flows with different granularity levels. For example, /24 subnets can be used. Packets from the same subnet will be aggregated together. Then, each flow will contain information about all traffic of the subnet it represents.

Čeleda et al. [22] also explain how packets are aggregated inside flow caches: tables of flows in flow exporters. When a packet whose flow key is not yet in memory is received, a new flow record is created. All packets with the same flow key received afterwards will cause an update of the corresponding flow. However, flow cannot wait for new packets indefinitely. Passive timeout is used for exporting a flow which did not receive a new packet for more than N seconds, usually between 120 seconds and 30 minutes [22]. Active timeout is used when flow keeps receiving new packets and would not be exported by passive timeout. Active timeout causes a flow to be exported after a maximum of M seconds even if new packets are still coming, usually between 15 seconds and 5 minutes [22]. The current flow is exported, and a new flow for the same key is created. Exported flows are sent to the flow collector using flow export protocols.

NetFlow is a flow export protocol developed by Cisco [24]. The two most used versions are NetFlow v5 (2002) and v9 (2004). Another flow export protocol is J-Flow by Juniper, which is mostly compatible with NetFlow v9. However, IP Flow Information Export (IPFIX) defined in RFC 7011 [26] is the only standardized protocol (2013), based on NetFlow v9 [24]. Trammell et al. [24] explain that IPFIX messages are defined via templates. Users can define their own data fields and highly customize the protocol. IPFIX message is shown on the Figure 2.2. IPFIX template defines what fields (Information Elements, IEs; defined by IANA) are contained in flow records [24]. Each flow record starts with the template ID it uses. Therefore, it is also possible to send messages with different formats [22].

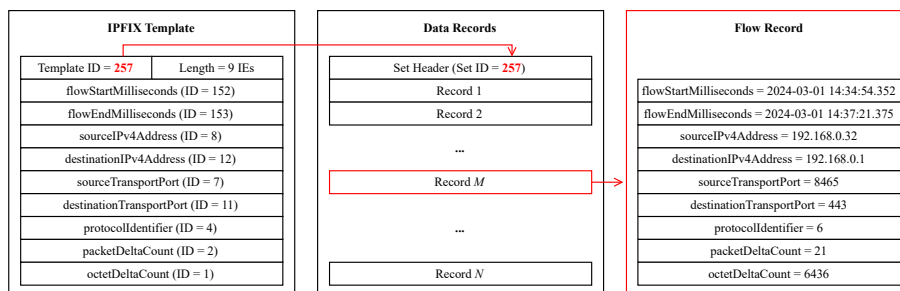


Figure 2.2: IPFIX messages

2.2.2 Flow Collector

Flow collector stores and pre-processes flow records, which are received from flow exporters [22]. It is possible that the collector receives data from more than one flow exporter. Čeleda et al. [22] write that after received flow records are pre-processed and stored, data analysis takes place.

Flow collector usually consists of a server and a process running on it. It has an open port for incoming flow data. Several flow collector implementations are available, such as open-source ipfixcol¹. Commercial flow collectors such as FlowMon Collector² are also available. Moreover, Cisco provides a physical device with flow collector capabilities³.

2.3 Network Traffic Analysis

Network traffic analysis is a process of examining network telemetry and trying to deduce as much information as possible [27]. It can be applied to both unencrypted and encrypted communication [28]. As mentioned before, most network traffic nowadays is encrypted; therefore, this method is essential for providing insight into what is going on in the computer network.

When it comes to traffic analysis, more information can be deduced from more data available. More available data also helps to create more accurate communication patterns and better situational awareness [29]. Since communication patterns usually differ based on the communication protocol used, traffic analysis can provide a traffic type label or used communication protocol for each analyzed flow record. Therefore, network traffic can be divided into groups based on similar traffic types and used protocols. This provides basic but essential insight into computer networks.

Once network traffic has been classified, network traffic models can be built. Such models can be used for the prediction of network load and for the detection of traffic loads that are much larger than expected: anomalous traffic. When anomalous traffic is detected, it can mean that a Denial-of-Service or data exfiltration is taking place. Therefore, it provides essential information about monitored computer networks. Moreover, characteristics of flows representing known attacks can be extracted and used for the detection of similar flows and other potential times when this attack happened.

2.3.1 Threat Detection

Threat detection monitors a network for policy violations, malicious activity, and potential security threats [30]. Once a threat is detected, an alert is created and either directly sent to the network operator or to a Security Information and Event Management (SIEM) system. The SIEM then enables operators to examine the alert in a wider context [31].

A possible setup for threat detection is shown on Figure 2.3. Network telemetry is processed by automatic threat detectors, which send alerts to its output with information about detected threats and security incidents [31]. Such alerts are collected by a SIEM system and stored in a database. Network security operators access this database, process and validate alerts and take appropriate action [31]. The action can vary from sending a warning email to a user or an institution, which causes the alert, to blocking ongoing communication of the guilty node.

¹<https://github.com/CESNET/ipfixcol2>

²<https://www.flowmon.com/en/products/appliances/netflow-collector>

³<https://www.cisco.com/c/en/us/support/security/stealthwatch-flow-collector-series/series.html>

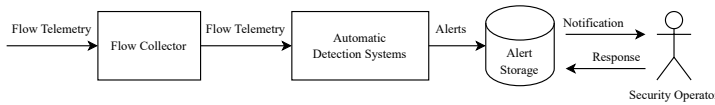


Figure 2.3: Threat detection pipeline

Threat detection mechanisms are divided into two groups. The first one is called signature-based threat detection [30]. Network traffic is searched for known patterns of malware and attacks, such as a sequence of bytes or a unique traffic type (for example, created by an SSH bruteforce attack). However, signature-based detection cannot detect any new attacks, only the ones we already know. The second group is called anomaly-based detection [30]. It can detect previously unknown attacks by modeling normal behavior and then performing detection of behavior that significantly differs from normal behavior. Methods used for network traffic analysis and threat detection are described in the Chapter 3.

2.3.2 NEMEA

Network Measurements Analysis (NEMEA) system was introduced by Čejka et al. in [32]. It is a system for online stream-wise network traffic analysis. As explained by Čejka et al. [32], the NEMEA system is composed of a number of independent interconnected modules. Every module serves a specific task, such as filtering, anomaly and threat detection, statistics, and more.

NEMEA works with network flows and is situated after the flow collector. Čejka et al. [32] explain that is a key difference when compared to other systems, such as Suricata⁴ and Snort⁵. Since these systems work with packets, they have more information available. On the other hand, NEMEA processes flow and can achieve better performance [32].

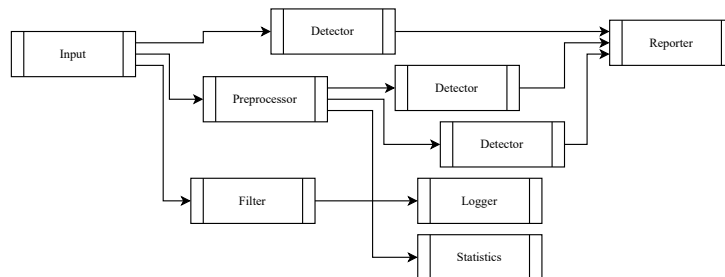


Figure 2.4: NEMEA System

Čejka et al. [32] demonstrate a possible deployment of a NEMEA system shown on Figure 2.4. Module interconnection is performed by unidirectional interfaces. A special binary format called *UniRec* is used for message transfer. As also explained by Čejka et al. [32], every deployed NEMEA system is unique since everyone can create a different interconnection and use and implement its own modules.

⁴<https://suricata.io>

⁵<https://snort.org>

Classification Methods

After network telemetry is captured, it has to be processed and analyzed. This chapter describes the most important methods for network traffic classification and threat detection. The basics of Machine Learning, data fusion and others are discussed. Furthermore, several (already implemented) software prototypes are discussed at the end of the chapter.

3.1 Pattern Matching

Pattern matching is a technique that searches supplied string data for a known *pattern*, it counts a number of occurrences of the pattern in data and discovers its positions [33]. For example, DPI uses pattern matching to look for known byte sequences and strings in packet payload to identify source applications [34].

The most basic pattern-matching technique is to match exact strings [33]. However, we can define more complex patterns to be searched for. Following pattern types were defined by Navarro in [33]:

Classes of characters The simplest possible pattern which is defined as a fixed-length string of text. We can describe every character by either the character itself or a group of characters, such as: `A[BC]D[0-9]`. For the pattern to be matched, the first character of the supplied string must be `A`, then either `B` or `C` must follow, then `D`, and then any character from range `0-9`, which is basically any digit character. Example strings where this pattern would be matched are `ABD1`, `ACD9`.

Optional and repeatable characters Previous pattern type can be extended by conditional occurrences of characters. This is indicated by the `?` following the symbol: `A?` means that `A` may be present in the corresponding position but is not required. This is useful, for example, when matching URLs: `https?://domain.com` can be used to match both HTTP and HTTPS traffic. Furthermore, multiple occurrences of characters can be permitted by using `+` (one or more occurrences) and `*` (zero or more occurrences), such as `A+C*D` will match strings, which start with at least one `A` character, followed by zero or more `C` characters, and end with the character `D`.

Bounded length gaps This type is used to match strings with specified lengths.

For example, $x(2,5)$ will be satisfied for strings with length from 2 to 5 characters. It is used when we need to denote distance: $Ax(2,5)B$ will discover strings which contain A and B patterns with 2-5 characters between them.

Regular expressions (REs) are complex patterns, which are typically used by commonly-used IDSs (Suricata, Snort) [35]. Regular expressions are recursive structures consisting of basic strings and union (denoted by $|$), concatenation (simply multiple RE put one after another), and repetition (denoted by $+$, $*$) of other RE. Operator precedence is as follows: repetition, concatenation, and union. However, parentheses $()$ can be used to change this order.

Network Expressions Special type of RE with excluded $*$ operator.

Regular expressions allow us to describe what patterns should be matched. However, it is also crucial to use effective pattern-matching algorithms, especially with nowadays constantly increasing loads of network traffic [36]. Gupta et al. [36] described several pattern-matching algorithms: Brute-Force Search and Rabin-Karp, Knuth-Morris-Pratt, and Boyer-Moore algorithms. A description of the chosen algorithms is provided below.

Brute-Force Search (also called Naive Search) is an algorithm which uses character-by-character matching. It requires no pre-processing and uses constant memory space. Naive search starts at the first character and tries to match the pattern. If the match fails, the algorithm moves to the next character and repeats the process. However, it is not efficient and suitable for practical deployment [36].

Rabin-Karp algorithm was introduced by Michael O. Rabin and Richard M. Karp in 1987 [37]. It uses hash values of the pattern and substrings with the same length as the length of the pattern. If the pattern hash and substring hash are not equal, hash of the next substring is calculated. If the hashes are equal, then a customized naive search is used to compare patterns and subsequences. However, Gupta et al. [36] found out that the practical use of this algorithm is highly dependent on the used hash function.

Knuth-Morris-Pratt algorithm was introduced by Donald E. Knuth, James H. Morris, and Vaughan R. Pratt in 1977 [38]. It is the first pattern-matching algorithm to achieve linear complexity. Knuth-Morris-Pratt algorithm uses prefixes: it matches prefixes of the pattern until the whole pattern is matched. Gupta et al. [36] assessed that this algorithm is efficient and can be used in practice.

Boyer-Moore algorithm was introduced by Robert S. Boyer and Strother J. Moore also in 1977 [39]. This algorithm searches for the index of the first occurrence in the text from right to left of the substring. It is considered the most efficient string-matching algorithm for many applications, such as IDSs and text editors [36]. As also assessed by Gupta et al. [36], the Boyer-Moore algorithm is suitable even for large text and pattern scenarios.

3.2 Machine Learning

A. L. Samuel was the first one to use the term Machine Learning (ML) in [40] in 1959. According to Samuel, ML is a field of study for making computers learn from data and experience to solve problems without explicit programming [40]. ML algorithms can be applied to various fields and tasks, such as computer vision, speech recognition, natural language processing, and more [41]. However, ML algorithms have to be trained before they can be applied and used to solve tasks. Moreover, ML can discover hidden data patterns which may be easily missed by humans [42]. Algorithms and models of ML are divided into several groups: supervised, unsupervised, semi-supervised, and ensemble learning [42].

The typical process of using ML is depicted on the Figure 3.1. Firstly, data pre-processing and dataset cleaning take place because the performance of Machine Learning highly depends on the used data, as known from the "garbage in, garbage out" principle (the term was coined by IBM programmer George Fuechsel) [43, 44]. It is followed by training the model and searching for the best hyperparameters (parameters of ML model; hyperparameter tuning). After a suitable model and its hyperparameters have been chosen, the model's performance is evaluated, and the model is ready to use.

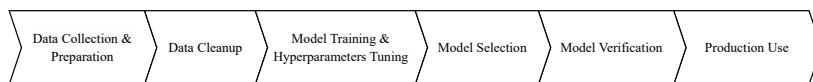


Figure 3.1: ML Model Development Process

Supervised ML uses a function which maps an input to an output [42]. This function is learned during the training phase and is used to deduce the output for previously unseen data. However, supervised models require external assistance [45]. Each element is represented by its feature vector. A class label for each element of the training dataset is required to infer the map function. Once the model is trained, it can produce class labels for previously unseen elements (their feature vectors). Supervised ML methods obtain a labeled dataset and can learn by example [46]. Moreover, Mahesh [42] writes that this dataset is split into *train* and *test* parts. The train part of the dataset is used to infer map function and model parameters. The test part of the dataset is used to assess the performance of the model: the model deduces class labels for elements which are then compared with the known and correct class labels from the original test dataset. Then, evaluation metrics can be calculated from these statistics.

Unsupervised ML does not use externally provided information nor a teacher and datasets used during the training phase contain no labeled data. On the other hand, large amounts of data are required to train unsupervised ML models [46]. These algorithms can discover hidden characteristics and patterns in data which would normally be almost impossible to find by humans. Unsupervised ML is typically used for clustering and association, anomaly detection, and autoencoding [46].

Semi-supervised ML is a combination of both supervised and unsupervised approaches [42]. It is useful when only part of the data is labeled. ML model is trained on the labeled data and can be used to label the rest of the data [42].

Ensemble learning is a type of learning where multiple ML models are used. This increases the performance of the model and reduces the likelihood of choosing a poor model [42]. Ensemble learning is based on the idea that "a group is stronger than an individual".

3.2.1 Machine Learning in Network Domain

Network traffic constantly evolves, as shown by Brabec et al. [47], and trained (and deployed) ML models can become obsolete in time [48]. This means that the accuracy of deployed ML models will be lower due to the appearance of new network protocols (or their new variants) in computer networks. Moreover, new application versions using the network differently will cause the same thing. Then, ML models can degrade over time, and training them with relevant data and keeping them up-to-date is essential. One of the possible solutions is an Active Learning (AL) approach. AL can manage the dataset on its own and quickly re-train the ML models.

Settles [49] explains that the key hypothesis of Active Learning (AL) is the following: "if the learning algorithm is allowed to choose the data from which it learns, it will perform better with less training" [49]. AL uses smart data querying to reduce the amount of labeled data needed and also deals with vast amounts of incoming data [50].

Pešek et al. [50] proposed Active Learning Framework (ALF): a modular framework with state-of-the-art methods of AL for continuous and autonomous dataset management and automatic ML model re-training. According to Pešek et al. [50], ALF can manage datasets and update deployed ML models to keep them as reliable and accurate as possible.

3.2.2 Description of Selected Supervised ML Models

Decision Tree

Decision Tree is a model of supervised learning. As explained by Quinlan in [51], it is a flow-chart-like structure (a tree graph), which starts in its root. Every internal node represents a condition (a test) for a value of a feature, such as `Number of packets >= 30` and each subtree represents the result of the condition [45]. Moreover, every leaf node represents a class label and the full decision path taken. One of the first algorithms for building a decision tree from data was designed by John Ross Quinlan in 1983 [52].

Random Forest

Random Forest is a model of ensemble learning approach that uses multiple decision trees [53]. It was introduced by Tin Kam Ho in 1995 [53]. As explained by Biau et al. [54], this model works on the "divide and conquer" principle: "sample fractions of the data, grow a randomized tree predictor on each small piece, then paste (aggregate) these predictors together" [54].

K-Nearest Neighbours

K-Nearest Neighbours (KNN) uses reasoning based on memories rather than other methods [55]. KNN was initially introduced by Evelyn Fix and Joseph

Lawson Hodges [56] in 1951. As explained by Stanfill et al. [55], the core concept is similarity-based induction. Moreover, KNN works directly with data instead of inferring rules for later decision-making. When asked to classify an element, KNN looks for the best match in its "memory" (training dataset). The final label is assigned based on the prevalent label of the K closest samples.

AdaBoost

Adaptive Boosting (AdaBoost) was introduced by Yoav Freund and Robert E. Schapire [57] in 1995. AdaBoost was the first practical boosting algorithm [58], based on the idea of creating highly accurate predictions by utilizing and combining many relatively weak and inaccurate rules.

3.2.3 Description of Selected Unsupervised ML Models

K-Means Clustering

K -Means Clustering is an algorithm of unsupervised learning, introduced by Stuart Lloyd in [59]. K -Means divides data into K groups based on their similarity. Naeem et al. [46] explain that the key is to find a centroid for each cluster. Then, an element belongs to any cluster whose centroid is the closest one.

Neural Network

Neural Network (NN) was initially proposed by Warren S. McCulloch and Walter Pitts [60] in 1943. Nowadays, a special type of NNs called autoencoders are part of unsupervised learning. General NNs are commonly referred to as self-supervised learning. Schmidhuber [61] explains that a neural network is a network of many simple processors (neurons). The first layer of neurons is activated by input sensors or data, and others are activated through weighted connections of previously activated neurons. Typical neuron activation function is shown on equation 3.1, where w_i is a weight of i -th input, x_i is a value of i -th input, and b is a bias. Neurons and their structures are inspired by animal and human brains: a nervous system formed by neurons, each having a certain threshold to initiate an impulse [60]. Neural networks can be used as unsupervised learning models, however, they can also be trained using backpropagation as a supervised learning model [61].

$$y = \sum_i w_i x_i + b \tag{3.1}$$

3.3 Anomaly Detection

Anomaly detection is a data analysis task which is used to find observations in datasets that do not conform to expected behavior. Hawkins defined an anomaly as "an observation which deviates so much from other observations as to raise suspicions that it was generated by a different mechanism" [62]. Anomalous observations are important because they represent rare but significant events [63]. For example, detection of anomalous traffic can mean that a computer was hacked or that there is an ongoing attack. Moreover, such events usually invoke critical actions to contain the attack. Ahmed et al. [63] defined several types of anomalies, described below.

Point Anomaly is an individual observation which significantly deviates from the rest of the data points [63]. For example, fuel consumption is usually around 5 liters per day but if it were 50 liters on any random day, it would be considered a point anomaly.

Contextual Anomaly is a type of anomaly which is caused by a particular context or when a certain condition is fulfilled [63]. For example, higher expenses on a credit card are caused by the Christmas period. It can be considered as normal behavior, but the same high expense during any other month would be considered anomalous.

Collective Anomaly is when a collection of similar observations behave as anomalies with respect to the rest of the dataset [63]. For example, many low values in the ECG (electrocardiogram) graph can mean an abnormal premature contradiction, but one low value in ECG would not be considered anomalous [63].

As pointed out by Chandola et al. [64], the output of the anomaly detection process can be either a score or a label. Usually, binary labels are used and observations are labeled as either *normal* or *anomalous* [64]. Scoring is useful when an analyst later wants to filter the reported anomalies. Moreover, analysts can prioritely respond to the anomalies with higher scores [64].

Classification-based Anomaly Detection

Classification-based anomaly detection relies on experts' knowledge [63]. Network operators provide characteristics about attacks to the detection system and similar attack patterns can be detected in the future [63]. However, such a system is vulnerable to new types of attacks. As explained by Ahmed et al. [63], classification-based detection builds a profile of normal traffic and detects anomalous traffic which deviates from the built profile. According to [63, 64], Support Vector Machine (SVM; a supervised ML model), Bayesian Network, Neural Networks, and rule-based techniques are used in classification-based anomaly detection process.

Nearest-Neighbour-based Anomaly Detection

According to Chandola et al. [64], nearest-neighbor-based anomaly detection is based on the assumption that "normal data instances occur in dense neighborhoods, while anomalies occur far from their closest neighbors" [64]. Such

methods require distance or similarity metrics which measure how close two data points are, for which Euclidean distance is a popular choice. For multivariate data points, distance is calculated for each element and then combined [64].

Clustering-based Anomaly Detection

Clustering-based anomaly detection uses unsupervised ML models to group similar data points into clusters [63, 64]. Moreover, several types of clustering can be used, as described below.

1. Normal data points belong to a cluster, anomalies are data points that do not belong to any cluster [64]
2. Normal data points lie close to the closest cluster centroid and anomalies lie far away [64]
3. Normal data points belong to large and dense clusters while anomalies form small or sparse clusters [64]

Statistical-based Anomaly Detection

Statistical-based anomaly detection fits statistical models to the given data and then uses statistical inference tests to check if unseen data points fit the model as well or not [64]. Data points with a low probability of being present in the statistical model are declared as anomalies. Statistical models can be divided into parametric (Gaussian-based models, χ^2 test) and non-parametric (histogram-based models) [64].

$$\begin{aligned}\hat{X}(t+T) &= \bar{X}(t) + \bar{b}(t)\left[\frac{1}{\alpha} + T - 1\right] & (3.2) \\ \bar{X}(t) &= \alpha X(t) + (1 - \alpha)\hat{X}(t-1) & 0 < \alpha < 1 \\ \bar{b}(t) &= \alpha[\hat{X}(t) - \hat{X}(t-1)] + (1 - \alpha)\hat{b}(t-1)\end{aligned}$$

Simple Exponential Smoothing (SES) is a widely used method for anomaly detection [65]. It was introduced by Robert G. Brown and Arthur D. Little in 1956 [66]. According to Brown [66], if values of a function $X(t)$ for times $t = 0, 1, \dots, t$ are known, an estimate of the future value at the time $t + T$ can be calculated as shown on equation 3.2. Parameter α defines the period over which the past values effectively contribute to the estimation of the next value. Small choices of α define long periods, therefore the average moves slowly and sudden peaks of values are missed. On the other hand, large values α create short periods and sudden peak values are more projected into the estimated values [66].

3.4 Data Fusion

Castanedo [67] defined data fusion as a combination of multiple data sources to obtain improved (higher quality, more relevant) information. Data fusion techniques are typically used in multisensor environments, with the main goal of

aggregating information from multiple data sources to obtain lower error probability and higher reliability [67]. Moreover, Castanedo [67] divides data fusion techniques into: (i) data association, (ii) state estimation, and (iii) decision fusion.

Data association problem is when a sensor provides measurements whose origins are uncertain; meaning not necessarily the target of interest [68]. The consequent problem is how to select measurements to be used to update the state of the target of interest. According to Bar-Shalom et al. [68], (Extended) Kalman Filter can be used to solve such problems. Moreover, minimum mean square error (MMSE) is used to estimate the target state and the associated uncertainty.

State estimation tries to determine the state of the target based on the provided observations and measurements [67]. However, observations can be either valid observations of the target or irrelevant noise. The problem is to determine a global state of the target from these observations [67], usually with the MMSE estimate [68]. According to Bar-Shalom et al. [68], this can be achieved by the pure MMSE approach which uses smoothing over all events. However, other approaches can be used as well, for example heuristics. Castanedo [67] states that Maximum Likelihood and Maximum Posterior approach, and Kalman Filter can be used as well.

Decision fusion makes a high-level inference about observations produced by sensors about targets of interest [67]. Castanedo [67] explains that a decision is taken based on knowledge of the observed situation. Moreover, the decision process must take uncertainties and constraints into consideration. Bayesian methods perform information fusion by combining evidence based on the Bayes rule, shown on equation 3.3. Moreover, Dempster-Shafer inference can be used [67].

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \quad (3.3)$$

3.4.1 Kalman Filter

Kalman filter was introduced by Rudolf E. Kálmán [69] in 1960. According to Welsch et al. [70], Kalman filter is a "set of mathematical equations that provides an efficient computational (recursive) solution of the least-squares method" [70]. It is a powerful method since it can estimate past, present, and future states.

As explained by Welch et al. [70], discrete Kalman filter estimates the state x of a process governed by the linear stochastic difference equation (3.4) with a measurement z (3.5). Matrix A relates to the state at the time k to the state at the time $k + 1$. Matrix B relates to the control input u to the state x . Matrix H relates to the state of the measurement z_k . Random variable w_k represents the process noise, v_k represents the measurement noise.

$$x_{k+1} = A_k x_k + B u_k + w_k \quad (3.4)$$

$$z_k = H_k x_k + v_k \quad (3.5)$$

3.4.2 Dempster-Shafer Theory

Dempster-Shafer Theory (DST) was initially introduced by Arthur P. Dempster [71] and later extended by Glenn Shafer [72]. DST is a mathematical theory of probability which generalizes the Bayesian theory. It can represent incomplete knowledge, updating beliefs, and it allows evidence combination and uncertainty representation [67].

The base concept is called the frame of discernment, which is defined in Lemma 1. Then, elements of the set 2^Θ are called hypotheses and based on evidence E , each hypothesis is assigned a probability by basic probability assignment or a mass function $m: 2^\Theta \rightarrow [0..1]$ satisfying $m(\emptyset) = 0$ and $m(2^\Theta) = 1$ (sum of probabilities of all hypotheses is 1).

Lemma 1 (Frame of Discernment) *Let $\Theta = \theta_1, \theta_2, \dots, \theta_N$ be the set of all possible states that define the system, and let Θ be exhaustive and mutually exclusive due to the system being only in one state $\theta_i \in \Theta$, where $1 \leq i \leq N$. The set Θ is called a frame of discernment because its elements are employed to discern the current state of the system [67].*

As explained by Castanedo [67], DST defines belief function $bel(H) : 2^\Theta \rightarrow [0..1]$ to express incomplete beliefs in hypothesis H as: $bel(H) = \sum_{A \subseteq H} m(A)$. Moreover, the doubt level of H can be expressed as: $doubt(H) = bel(\neg H)$.

In addition, DST defines plausibility function $pl(H) : 2^\Theta \rightarrow [0..1]$ to express a plausibility of each hypothesis H as: $pl(H) = 1 - doubt(H) = \sum_{A \cap H = \emptyset} m(A)$.

Finally, DST provides a method to combine two mass probability functions m_1 and m_2 by the following mechanism, known as the Dempster's Rule of Combination [67]:

$$m_1 \oplus m_2 = \frac{\sum_{X \cap Y = H} m_1(X)m_2(Y)}{1 - \sum_{X \cap Y = \emptyset} m_1(X)m_2(Y)}$$

3.5 BOTA

Botnet Analysis (BOTA) system was introduced by Uhřiček et al. in [73]. BOTA targets high accuracy and explainability by utilizing weak indicators and heterogeneous meta-classifiers. Moreover, BOTA targets high-speed networks achieving high efficiency and can potentially protect millions of devices [73].

As described by Uhřiček et al. [73], the classification pipeline has two levels: (i) weak indicators and (ii) heterogeneous meta-classifiers. Each weak indicator is processed by its associated detector: Command & Control communication detector, anomaly detectors, DHT, Tor, and Stratum detectors. Then, the results of the listed detectors are combined together by the functions listed in (3.6), (3.7), and (3.8). A_b is the anomaly in total bytes sent, A_p is the anomaly of total packets sent, A_{dip} is the anomaly in the used number of destination IP addresses, and finally A_{dp} is the anomaly in the used number of destination ports.

$$F_1 = C\&C \wedge (A_b \vee A_p \vee A_{dip} \vee A_{dp}) \quad (3.6)$$

$$F_2 = Tor \wedge (A_b \vee A_p \vee A_{dip} \vee A_{dp}) \quad (3.7)$$

$$F_3 = DHT \wedge Stratum \quad (3.8)$$

Combination function F_1 (equation 3.6) detects devices infected with Mirai, Gafgyt, and Tsunami botnets. Combination function F_2 (equation 3.7) detects C&C communication hidden in Tor. Finally, combination function F_3 (equation 3.8) targets peer-to-peer communicating cryptocurrency miners.

Uhříček et al. [73] proposed a C&C communication detector as a Machine-Learning-based classifier which utilizes the AdaBoost model. DHT detector is a simple signature-based detector that looks for `d1:ad2:id20:` pattern, which reveals BitTorrent’s DHT communication. Stratum detector detects Stratum mining protocol also by pattern-matching. Tor detector utilizes REST API provided by The Tor Project and detects communication with Tor relay nodes by simple IP blocklist. Anomaly detectors of the number of packets sent, the number of bytes sent, the number of unique destination IP addresses, and finally the number of unique destination ports are based on Brown’s simple exponential smoothing algorithm.

BOTA system works in 5-minute time windows [73], meaning that the described combination functions are invoked once every 5 minutes. After the combination, each detector is reset. Moreover, anomaly detectors work in one-minute windows meaning that for every tick of the whole BOTA system, the final result of each anomaly detector is composed of 5 sub-results. For a final result to be positive, the anomaly must be detected in at least two consecutive one-minute windows [73].

3.6 DeCrypto

DeCrypto is a system for detection of cryptocurrency miners, presented in [74]. It uses a similar weak-indicators principle as BOTA [73], to achieve more reliable results and lower false-positive rates. Moreover, Dempster-Shafer theory is used for data fusion to further lower the false-positive rate and to increase its reliability [74].

As also explained in [74], multiple *weak-indication* classifiers are used: Stratum detector, TLS SNI classifier, and ML classifier. DeCrypto system works on a per-flow basis meaning each flow is supplied to all detectors, and their output is processed by the Meta classifier. Stratum detector performs pattern-matching and detects flows representing unencrypted Stratum (cryptocurrency mining protocol) communication. TLS SNI classifier also performs pattern-matching and detects flows representing encrypted communication with mining and suspicious keywords in TLS SNI. Lastly, the ML classifier performs statistical-based detection of flows with the same characteristics as cryptomining communication. Meta classifier takes the output of all the previously described detectors, performs data fusion with the use of DST and provides more reliable detection results.

3.7 VPN Detection

Čtrnáctý proposed the detection of OpenVPN communication based on finite automata in [75]. Packets are parsed based on known OpenVPN communication protocol. Moreover, Čtrnáctý [75] proposed finite-automaton detection for Cisco AnyConnect VPN. The automaton for OpenVPN is shown on the

Figure 3.2. However, proposed automatons can be used only for the specific VPN protocol.

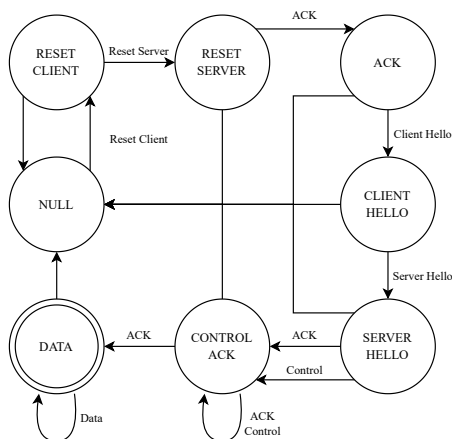


Figure 3.2: Finite automaton for OpenVPN detection

Jiráček proposed an algorithm for general detection of VPN protocols in [76]. The algorithm looks for packet patterns with the following TCP flags: SYN, SYN-ACK, ACK; also called SSA sequence. The discovered SSA sequence of the TCP handshake in UDP traffic is highly suspicious. According to Jiráček [76], it can be used for the detection of encapsulated packets and VPN.

Valach proposed the detection of WireGuard VPN in [77] based on DPI and Machine Learning. The DPI approach scans packet contents and tries to parse WireGuard protocol fields. On the other hand, the ML approach uses packet features to predict if a flow represents WireGuard communication or not. AdaBoost and LightGBM were used and both achieved similar results: $F1$ score between 90-91% [77].

Weak Indication Framework

After reviewing the state-of-the-art methods for network traffic classification, we can design and define what the Weak Indication Framework should contain and how it should work. This chapter describes how the WIF will look and then all of its parts and features together with examples directly from the code.

4.1 Introduction

We noticed that more complex detection methods usually consist of multiple detectors, based on our analysis in Chapter 3. Moreover, we can see that many principles are used repeatedly. Therefore, we decided to implement the most frequently used methods for network traffic classification as a library called Weak Indication Framework (WIF). Then, everyone can develop new detectors and mechanisms without re-implementing already prepared methods. Rather than that, new and more complex methods can be implemented.

Furthermore, some detectors use multiple methods (or indicators). The final prediction is obtained by combining the results of sub-detectors. Detection methods designed in this manner provide more accurate results with higher explainability. Moreover, it is becoming more common to design methods in this way. Therefore, WIF will be designed to support such architectures.

Weak Indication Framework will contain several types of objects for different purposes. This way, each group of classes will have the same interface, and once a developer knows how to use one, he will also know how to use other types from the same group. Therefore, the time needed to learn and use WIF will be shorter than if every provided class had a different interface and use case. The groups are shown on Figure 4.1 and are as follows: *classifiers* (implementation of classification methods), *combinators* (implementing data fusion algorithms), and *reporters* (for reporting additional information about the detection methods). It will be possible to chain classifiers and combinators one after another. This modular architecture will allow researchers and developers to quickly develop new and more advanced detection methods based on the ones regularly used. Moreover, it will support the architectures using multiple indicators and sub-detectors. Finally, high explainability can be achieved with correctly-used reporters.

Since WIF aims to provide the most used methods for network traffic classification, it must contain a possibility to use Machine Learning. Machine

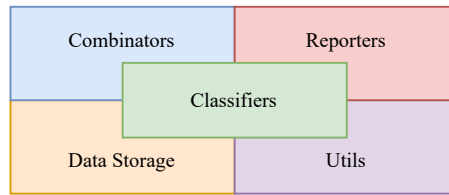


Figure 4.1: High-level view of WIF object groups

Learning is widely used nowadays in network traffic classification and threat detection, especially scikit-learn library⁶. It is a Python library that is arguably the most commonly used by researchers worldwide. The majority of ML models are trained and manufactured by the scikit-learn library. WIF will contain functionality to work with this library and models, enabling its users to utilize ML models from basically any researcher in the domain.

Another essential functionality that must be contained in WIF is pattern matching. Pattern matching is widely used for searching for specially crafted payloads, ill-formed packets, suspicious keywords indicating malicious activity, and more. It can detect known byte sequences used in SQL injections, buffer-overflow attacks, and other threats. Moreover, pattern matching can discover known protocol structures and their identifying magic values. WIF will contain functionality for matching both plain strings (keywords) and fully-featured regular expressions for advanced use cases.

Last but not least, functionality for blacklist-based detection must be present. Despite being one of the oldest and most basic methods, blocklists are still widely used. IP blacklist lookup must be efficient since blocklists can contain thousands of IP addresses to ensure high throughput of the WIF library. Moreover, exact matching is not always the wanted option. Another helpful functionality is checking if an IP address belongs to a (blocklisted) IP subnet. WIF will contain both exact checking and checking for being a member of a particular IP subnet.

Even though WIF is primarily designed for network traffic classification, threat detection, and building detection modules, it can also be used in other scenarios. For example, information fusion plugins for Kibana and ELK can also benefit from the WIF-provided functionalities. However, such use cases are out of the scope of this thesis and are only mentioned as an example of the possible multidisciplinary use of the WIF library.

⁶<https://scikit-learn.org>

4.2 Development Environment & Process

Git and GitLab are used as version control tools and source file storage. The dir tree below describes the repository structure.

```

├── ansible/.....ansible for preparing the CI environment
├── ci/.....CI jobs definition
├── cmake/.....folder with cmake modules
├── docker/.....folder containing the Docker image for CI
├── include/.....include folder
│   └── wif/.....WIF header files
├── pkg/.....files needed for creating WIF RPM packages
├── src/.....folder with source files
│   ├── external/.....links for external libraries
│   └── wif/.....folder with WIF source files
├── .clang-format.....clang-format configuration
├── .clang-tidy.....clang-tidy configuration
├── .editor-config.....editor-config configuration
├── .gitignore.....definition of ignored files by git
├── .gitlab-ci.yml.....definition of Gitlab's CI jobs
├── CMakeLists.txt.....the main CMake file
├── Doxyfile.....Doxygen configuration file
├── LICENSE.....file with license
├── Makefile.....Makefile
├── Pipfile.....Python dependencies configuration
├── Pipfile.lock.....Python dependencies configuration
└── README.md.....main readme file

```

CMake is used to manage the build system and the project's dependencies. Two compilation modes are defined: **Debug** and **Release**. The debug mode appends the `-g` flag to the compilation options, causing the compiler to put additional debug symbols into the output binary. On the other hand, the release mode instructs the compiler to perform the best possible optimizations and targets speed and performance.

In addition, we defined an argument called `BUILD_WITH_UNIREC` in CMake. When this option is enabled, source files that depend on the `libunirec` and `libunirec++` are compiled and added to the library `.so` file. Then, the WIF library contains additional classes, which will be described later. Otherwise, they are not part of the library.

Furthermore, the repository contains `rpm` folder with everything needed to create an RPM package with the WIF library. It can be created by calling `make rpm` in the repository's root. Then, a tar file with sources will be created together with an RPM package with sources. In addition, target installation folders can be set via CMake. By default, the WIF `.so` file is installed in `/usr/lib64/` and header files into `/usr/include/wif`.

Last but not least, CMake is used for library versioning. The WIF library is attached to this thesis in `LIBWIF v3.1.1`. The Doxygen⁷ utility is used to generate documentation from the source files automatically.

⁷<https://www.doxygen.nl>

CI/CD on the GitLab server performs several checks to keep the code in high quality. Firstly, compliance with the defined code style is checked via clang-format and editorconfig tools. The alphabetical order of the included clauses (`#include <headerFile>`) is also checked. Then, the clang-tidy linter evaluates the committed code and diagnoses typical programming errors and inefficiencies. If all previous checks were passed, the follow-up job tries to build the library. The last performed job verifies that the RPM build is set up correctly and that the RPM package can be created and installed.

Development of new functionalities takes place in separate branches. The `main` branch is marked as `protected`, meaning that no changes can be committed directly to it. Before any branch is merged to the `main` branch, a review must be performed and review notes must be integrated into the proposed changes. In addition, all CI tests must be passed. The reviews were conducted by Ing. Pavel Šiška — a very experienced C++ developer from CESNET, for which he deserves a big thanks. His review notes made source codes be written in a very clear and readable way and took the WIF software to a professional level.

4.3 Architecture

Weak Indication Framework uses a modular architecture. It consists of several parts that group together objects with similar purposes. Objects of the main groups have the same interface. Then, we can implement new functionalities while keeping the previously defined and well-known interface of each group, simplifying the usage of our library.

The first part called *storage* will contain structures for data representation in memory. Therefore, WIF will be able to process any provided data. We focus on the flow data, but no limitation is defined; theoretically, it can also represent other types. The object representing one flow in memory is called *FlowFeatures*. Objects in this group share a common purpose (data representation in memory) rather than a common interface.

The second and probably the most crucial part of the WIF library contains `classifiers`. Classifiers are objects that implement classification and detection methods. They perform an actual classification of *FlowFeatures* objects. Moreover, classifiers share a common interface. The common interface will provide methods for the classification of a single flow or a burst of flows. Selected methods from the Chapter 3 will be implemented, each as its own classifier.

The third part consists of *combinators*. Combinators are similar to classifiers but implement data fusion and combination methods rather than classification methods. Combinators are also an essential part of the WIF library because they allow the building of detection methods using multiple indications and sub-detectors. Combinators also share a common interface.

For example, passing the combined output of several classifiers into another classifier will be possible, creating more complex but more accurate detection methods. Therefore, advanced detection methods can be easily implemented using multiple classifiers and combinators, following the modular architecture of the WIF library.

The fourth part will contain *reporters*. Understanding how an advanced detection method (consisting of several classifiers and combinators) produces

the final results can be non-trivial. Reporter objects can be used to exfiltrate internal states, sub-results, and additional data to support high explainability. Exfiltered data can be sent to other processes (ML features to Active Learning Framework, for example) or stored in a database. A network operator responding to a threat can then use stored data and logs to further investigate and better understand what is happening.

The final part of the library is called `utils` because it contains several helper classes that other parts of the library require. However, they are made available to the WIF users. These classes do not share a common interface. For example, the `utils` part contains a class called `Timer`, which can be used to run a code in another thread periodically.

A possible detection module built on the top of the WIF library is depicted in the Figure 4.2. Any number of classifiers, combinators, and reporters can be used and chained together in any way the developer wants. Moreover, adding and expanding WIF-based modules will be easy because new classifiers can be easily added to the existing WIF-based module. For example, adding support for the newer protocol version to an existing detector of this protocol will be non-demanding.

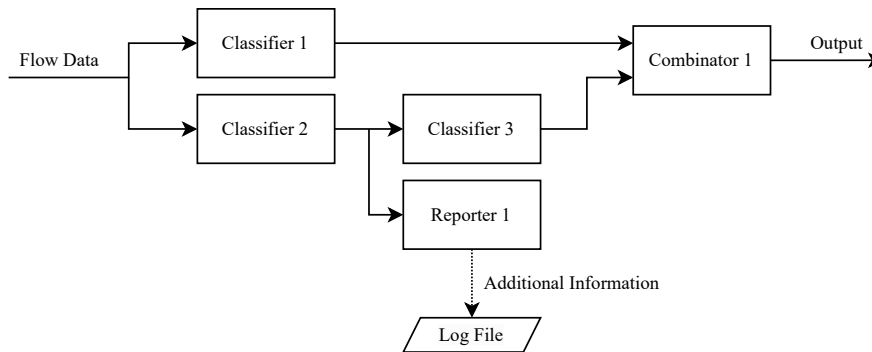


Figure 4.2: Example of WIF-based detection module

Moreover, WIF will be implemented in C++ to achieve high efficiency so that WIF-based classifiers can be deployed to high-speed national-level networks as part of the NEMEA system. Despite that the WIF library is designed to be independent of NEMEA libraries: `libtrap` and `libunirec(++)`. However, additional features will be available in the WIF library when compiled with `BUILD_WITH_UNIREC` option.

The intended use of WIF is depicted on the Figure 4.3. Firstly, a WIF-based detector receives flow data and transforms it into a data representation from WIF: `FlowFeatures`. Then, the used classifiers, combinators, and reporters from WIF are invoked, and the detection is performed. Finally, an alert or an output message is created and sent to the detector's output. We intend to develop WIF-based modules as modules of the NEMEA framework (described in Subsection 2.3.2). Therefore, the flows are received via the input `unirec` interface, and alerts are sent to the output `unirec` interface. However, it is not required, and other architecture and data sources can be used. For example, data can be retrieved from the database, and alerts can be sent to an unix socket. Developers of the WIF-based detectors are free to use what they want.

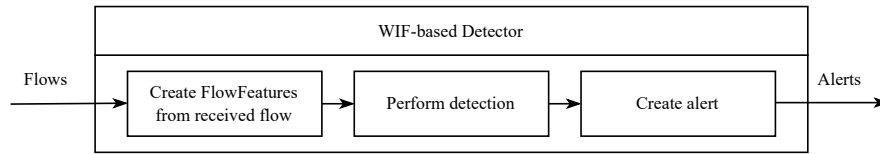


Figure 4.3: Intended use of the Weak Indication Framework

4.4 Data Structures

The WIF library contains its own data structures for internal data representation. The primary type is called `DataVariant`. It is a wrapper over `std::variant` with allowed types. The supported types and their possible use are described in the Table 4.1. It is designed to hold any value from flow telemetry.

Table 4.1: Supported data types by `DataVariant` class

Data type	Possible use case
<code>uint8_t</code>	TCP Flags, Transport layer protocol
<code>uint16_t</code>	SRC and DST ports, packet length
<code>uint32_t</code>	Transferred packet count, Burst information
<code>uint64_t</code>	Transferred byte count, flow ID
<code>double</code>	ML features (ratio, average size, ...)
<code>std::string</code>	TLS SNI, packet content
<code>IpAddress</code>	SRC and DST IP addresses
<code>std::vector<double></code>	Feature vector, probability array

A class called `FlowFeatures` represents a flow in memory. It holds a `std::vector` of `DataVariant` types pre-allocated to the specified number of features (provided as the constructor's parameter). This allows the object to allocate memory only once at the beginning without reallocating the vector later. Moreover, the `FlowFeatures` object can be cleared and re-used to represent another flow without allocating new memory. This is handy when the detector runs in a loop: a flow is received, detection is performed, and an alert is sent. The same `FlowFeatures` object can be used all over again, making the detector more efficient. Furthermore, a buffer (a vector) of `FlowFeatures` objects can be created and re-used.

Data stored in the `FlowFeatures` object can be accessed, set via templated `get()`, `set()` methods respectively. `FeatureID` type is a wrapped `uint16_t` and is used as an index type of the `FlowFeatures`. A value of type `T` stored on the certain `FeatureID` index can be obtained by calling `FlowFeatures::get<T>(FeatureID)`. Similarly, a value of type `T` can be set to the specified `FeatureID` index by calling `FlowFeatures::set<T>(FeatureID, T)`. Type `T` must be supported by the `DataVariant` class. Full `FlowFeatures` interface is shown on listing 1.

In addition, WIF contains a class called `IpAddress`. It can store both IPv4 and IPv6 addresses and can be stored in the `FlowFeatures` object. The interface of the `IpAddress` object is shown on listing 2. The object can be constructed from a single `uint32_t` type (in the case of IPv4; line 7 on listing 1).

```

1 // Note: virtual keywords, std namespace,
2 //       and parameter names are omitted
3 //       for increased code readability
4 using FeatureID = uint16_t;
5 class FlowFeatures {
6 public:
7     // Construct new object and pre-allocate memory for
8     //   ↪ featuresCount features
9     FlowFeatures(size_t featuresCount);
10    // Get feature of type T from index featureID
11    template<typename T>
12    const T& get(FeatureID featureID) const;
13    // Get DataVariant from index featureID
14    const DataVariant& getRaw(FeatureID featureID) const;
15    // Get feature on index featureID to newValue of type T
16    template<typename T>
17    void set(FeatureID featureID, T newValue);
18    // Get number of features of this flow
19    size_t size() const;
20    // Clear the flow features object
21    void clear();
22 };

```

Listing 1: Interface of the `FlowFeatures` class

It can also be constructed from bytes with specified IP version and endianness (the last parameter with the name `isLittleEndian`; line 8). Then, the class contains all sorts of getter methods for the IP version and underlying data. However, several basic methods were omitted to make the listing more legible.

Furthermore, the `IpAddress` class contains methods for comparison, sorting, and bitwise operations. The operator `less` is implemented so that every IPv4 address is before any IPv6 address. If IP versions are equal, operator `<` is called for bytes on the first position where they differ. The logical *and* and negation operators can be used to check if the IP address is a member of a particular IP subnet.

Lastly, the storage part of the WIF library contains `ClfResult` class (shortened from the Classification Result). This object represents a result of the classification operation of the classifier objects (discussed in Subsection 4.5): a return type of the `Classifier::classify()` methods. `ClfResult` object can hold two data types: either a value of a type `double` or a `std::vector<double>`. Sometimes, the classification result is 0 or 1 (boolean value) or a value representing a score (decimal number). For example, the result of a blacklist-based detection can be 0, 1 representing that the IP address was not found, found on the blacklist respectively. However, the result of the ML classification can be an array of probabilities for each known class. Therefore, the type used as a result must be able to represent both value types. The author of each classifier must define in the documentation what the `ClfResult` holds.

```
1 // Note: virtual keywords, std namespace,
2 //       and parameter names are omitted
3 //       for increased code readability
4 class IPAddress {
5 public:
6     enum class IpVersion { V4, V6 };
7     IPAddress(uint32_t);
8     IPAddress(const uint8_t*, IpVersion, bool);
9     IPAddress(const std::string&);
10    const uint8_t* data() const noexcept;
11    uint32_t v4AsInt() const noexcept;
12    const char* v4AsBytes() const noexcept;
13    const uint32_t* v6AsIntArray() const noexcept;
14    const char* v6AsBytes() const noexcept;
15    std::string toString() const;
16    friend bool operator==(...);
17    friend IPAddress operator&(...);
18    friend IPAddress operator~(...);
19    friend bool operator<(...);
20 };
```

Listing 2: Interface of `IPAddress` class, shortened.

4.5 Classifiers

Classifiers are the most important part of the WIF library. Each classifier provides a particular classification method. Moreover, all classifiers have a common interface defined by an abstract class called `Classifier`. This interface is shown on listing 3.

Every classifier is a child of the `Classifier` class, inherits its interface and implements the abstract methods. As we can see on lines 12 and 14 of the listing 3, methods for classification of a single flow and burst of flows must be implemented. In some cases, it may be more efficient to work with a burst of flows; see description of `ScikitMLClassifier` below. On the other hand, the efficiency of some classifiers is not affected, and single-flow classification can be called in a loop for each flow in the processed burst; see description of `RegexClassifier` below as an example.

Before any `classify()` method can be called, IDs of the source features must be set. This is done by calling `setSourceFeatureIDs()`, line 11 in listing 3, and providing a `std::vector` of indexes (`FeatureID` values) into the `FlowFeatures` object. Whole `FlowFeatures` is sent for classification, but the classifier only works with the `DataVariant` objects from the previously set source feature IDs. Furthermore, each classifier can require additional methods to be called before the first classification can be done. After the classifier object has been set up, `classify()` methods can be called repeatedly. The intended use is that the classifier is firstly set up, and then `classify()` methods are called in a loop for each received flow or burst of received flows.

Following classifier specializations were chosen based on the discussion with thesis supervisors and the review provided in Chapter 3: `IpPrefixClassifier`, `RegexClassifier`, `ScikitMLClassifier`, and `AlfClassifier`.

```

1 // Note: virtual keywords, std namespace,
2 //       and parameter names are omitted
3 //       for increased code readability
4
5 // Abstract class defining the interface of Classifier objects
6 class Classifier {
7 public:
8     // Getter for source IDs
9     const vector<FeatureID>& getSourceFeatureIDs() const;
10    // Setter for source IDs
11    void setFeatureSourceIDs(const vector<FeatureID>&);
12    // Perform classification for a single flow
13    ClfResult classify(const FlowFeatures&) = 0;
14    // Perform classification for multiple flows
15    vector<ClfResult> classify(const vector<FlowFeatures>&) = 0;
16 };

```

Listing 3: Common interface of the `Classifier` objects

4.5.1 IP Prefix Classifier

`IpPrefixClassifier` performs blacklist-based classification. The source feature IDs must point to the values of type `WIF::IpAddress` in the provided `FlowFeatures` instance. The resulting double value is set to either 0 or 1 depending on if at least one source feature contained an IP address on the blacklist. The used blacklist is obtained via the classifier’s constructor but can also be periodically set.

A utils class called `IpPrefix` represents a blacklist element. It contains an `IpAddress` object and `size_t` representing the number of bits in the prefix. An exact match will be performed when the number of bits is set to 32 (IPv4) or 128 (IPv6). The `IpPrefix` object can be constructed by either supplying the `WIF::IpAddress` object and then the number of bits is set so an exact match will be performed. Another option is to supply either a `std::string` or `IpAddress` and manually set the number of bits in the prefix. Both exact matching and checking if an IP address belongs to a particular IP range can be performed.

The provided blacklist is sorted before the actual classification can take place. The sorting order is set in a way that all elements of type IPv4 are before any IPv6 element. Then, a comparison of prefixes is performed. In a case that the prefixes match, the number of prefix bits is used. After the blacklist has been sorted, `std::binary_search` is used for an effective lookup of an IP address on the blacklist. The classifier supports IP prefixes with any prefix length. Full prefix length of IPv4 (32 bits) or IPv6 (128 bits) can be used for exact matching.

Another utils class `FileModificationChecker` can be used to detect a new version of the source blocklist on disk. This class can detect a new version of a file on disk by reading the last modification time. Periodical checks for new versions can be performed by utils class called `Timer` and loading of a new blocklist when the file change is detected.

4.5.2 Regex Classifier

`RegexClassifier` is a classifier performing pattern matching. The required type of fields in the `FlowFeatures` object for this classifier is `std::string`. The classification result for each flow is a `double` value. However, the resulting value depends on the selected operational mode of the regex pattern described below.

`RegexPattern` class represents what and how it is supposed to be matched. This class takes a vector of strings representing multiple regex patterns, which will be matched. Elements of the supplied vectors can be full regex patterns or keywords (plain strings). Then, character-by-character matching will take place. The second parameter is an operational mode, described below.

ALL All provided patterns have to be matched. The resulting double value is `REGEX_PATTERN_FOUND` (with a value 100.0) if all patterns were matched, otherwise `REGEX_PATTERN_NOT_FOUND` is returned with the value 0.0.

PART Matching of every pattern takes place. The resulting value is a percentage of the successfully matched patterns. The value ranges from 0.0 up to 100.0.

ANY If at least one pattern was matched, `REGEX_PATTERN_FOUND` (with a value 100.0) is returned. Otherwise `REGEX_PATTERN_NOT_FOUND` with value 0.0 is returned.

Regex Classifier has a `RegexPattern` object as the first parameter defining what patterns and in which mode the matching will be performed. The second parameter is a combinator object, used when multiple source features are used. For example, if two source feature fields are set, the provided combinator will be used to combine the pattern-matching results of each field to provide one final result.

The performance of the single-flow and multiple-flow classifications are equal since no demanding communication is needed, as in the case of the Scikit ML Classifier. However, we found out during testing that the regex implementation in the standard C++ library called `std::regex` is very slow. Therefore, boost regex library⁸ is used internally by the Regex Classifier.

We used DeCrypto datasets [74] and implemented a simple NEMEA and WIF-based module with two Regex classifiers. The module receives a flow and performs Stratum protocol detection using two regex patterns from the original DeCrypto system (described in Section 3.6). The program was run 100 times for each regex backend: `std::regex` and `boost::regex`. The data used for the experiment contained 2 250 000 flows. The Table 4.2 shows runtime statistics from our experiment. We can see that the `boost::regex` is almost ten times faster than the regex implementation from the standard C++ library.

⁸https://www.boost.org/doc/libs/1_78_0/libs/regex/doc/html/index.html

Table 4.2: Performance of DeCrypto systems

Regex backend	Seconds			
	Min [s]	Average [s]	Median [s]	Max [s]
<code>std::regex</code>	36.67	39.40	38.34	45.10
<code>boost::regex</code>	3.91	3.99	3.92	4.43

4.5.3 Scikit ML Classifier

`ScikitMLClassifier` is a classifier performing Machine-Learning-based classification via Python's scikit-learn library. Its source features are features of the used ML model, and the result of its single-flow classification is a vector of doubles, representing the output of the `predict_proba()` method called on the scikit-learn model. It is presumed that all ML features are doubles. Therefore, every index to `FlowFeatures` object must point to the `double` value. The constructor of this classifier has two parameters: a string with a path to the ML model in the pickle format⁹ and a path to the so-called bridge, described below.

```

1 import pickle
2
3 def init(modelPath):
4     with open(modelPath, 'rb') as f:
5         return pickle.load(f)
6
7 def classify(classifier, features):
8     return classifier.predict_proba(features).tolist()

```

Listing 4: Bridge module used by Scikit ML Classifier

Since scikit-learn is a Python library, Python C API¹⁰ is used together with a simple Python script called *bridge* for communication with the scikit library. The bridge module shown on listing 4 has to contain two functions: `init()` and `classify()`. The `init()` function obtains the ML model path, loads the pickle file, and returns the ready-to-use scikit ML model. The `classify()` function receives the scikit ML model and a 2D list of features. It internally calls the `predict_proba()` method and returns a 2D array of the resulting probabilities. These probability arrays are then available in the resulting `ClfResult` object for each flow.

It is highly recommended that this classifier only be used for processing bursts of flows since the Python scikit-library has to be called internally by the Scikit ML Classifier. Severe performance issues may arise when a single-flow `classify()` function is used. However, processing of bursts with at least 50 000 flows achieves sufficient speed. Single-flow classification performance issues are balanced by allowing the usage of ML models from the scikit-learn library, arguably the most frequently used library for Machine Learning.

⁹<https://docs.python.org/3/library/pickle.html>

¹⁰<https://docs.python.org/3/c-api/index.html>

4.5.4 ALF Classifier

`AlfClassifier` serves as a wrapper over Scikit ML Classifier. It provides interconnection with the Active Learning Framework (ALF, described in Subsection 3.2.1). This way, ALF can re-train ML models and WIF-based module can re-load them from the disk, preventing model degradation over time.

ALF requires several data to work correctly. The `AlfClassifier` must send ML features and output probability array from the ML model. Moreover, WIF will send a timestamp of the last model load. This is important because ALF can distinguish if the WIF-based module already uses the up-to-date model or if the incoming data are from the older version.

`AlfClassifier` class takes two arguments in its constructor. The first is an instance of the `ScikitMLClassifier`, which is then used internally for the classification. The second argument is a C++ reference type to `UnirecReporter` (described in Section 4.7), which forwards features and probability arrays to the ALF via `libtrap` and `libunirec` libraries. Since the `UnirecReporter` is used, it is only available if WIF was compiled with the `unirec` and `trap` support (`BUILD_WITH_UNIREC` enabled).

Moreover, `AlfClassifier` uses `Timer` and `FileModificationChecker` to check if the source ML model file was changed periodically. Once this event occurs, the ML model reload is performed during the next `classify()` call: it cannot be reloaded directly because the load operation must be performed by the same thread that initialized Python C API.

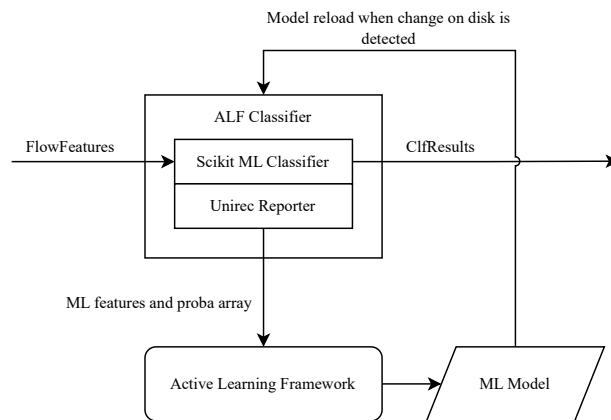


Figure 4.4: WIF-ALF interconnection

4.6 Combinators

Combinators are classes that provide methods for data fusion. Similarly, as classifiers, the common combinator interface is defined by abstract class `Combinator`, shown on listing 5. Combinators are much simpler than classifier objects. The primary method is called `combine()`, and it takes a vector of double values and returns a double with the fusion result.

```

1 // Note: virtual keywords, std namespace,
2 //       and parameter names are omitted
3 //       for increased code readability
4
5 // Abstract class defining the interface of Combinator objects
6 class Combinator {
7 public:
8     virtual double combine(const vector<double>&) = 0;
9 };

```

Listing 5: Common interface of the `Combinator` objects

4.6.1 Basic Combinators

Weak Indication Framework contains several combinators with basic functions. The first one is called `AverageCombinator` and provides a simple average operation. The second one is called `SumCombinator` and provides a sum operation.

4.6.2 Binary DST Combinator

`BinaryDSTCombinator` provides a data fusion operation based on the Dempster-Shafer Theory, described in Section 3.

Every input value is treated as a witness of an event, and each value is used to define one basic probability assignment function (hence *binary*). The value is used as a probability for the positive hypothesis and $1.0 - \text{value}$ as a probability for the opposing hypothesis. All basic probability assignment functions are combined together, and the resulting probability for the positive hypothesis is returned.

4.7 Reporters

Reporters are used to exfiltrate additional information from detection modules. They can help to understand what is going on during the development and research of new modules, as well as to help monitor deployed modules and provide inside information. Reporters can boost the overall explainability. Moreover, they can be used for active learning detection approaches.

The common interface is defined by an abstract class called `Reporter`, shown on listing 6.

Reporters work differently than classifiers and combinators. Their function resembles a finite state automaton. Firstly, `onRecordStart()` must be called to indicate the start of the new message. Then, the `report()` method can be called multiple times to supply data for the current message. The `DataVariant` type is the same variant used by `FlowFeatures`. Therefore, any supported type by `FlowFeatures` can be reported. After all the data was supplied, `onRecordEnd()` must be called to indicate the end of the current message. However, it does not mean the message is sent immediately: reporters might use buffering. The `flush()` method can be used to send messages immediately.

```
1 // Note: virtual keywords, std namespace,  
2 //       and parameter names are omitted  
3 //       for increased code readability  
4  
5 // Abstract class defining the interface of Reporter objects  
6 class Reporter {  
7 public:  
8     void onRecordStart() = 0;  
9     void report(const DataVariant&) = 0;  
10    void onRecordEnd() = 0;  
11    void flush() = 0;  
12 };
```

Listing 6: Common interface of the Reporter objects

4.7.1 Unirec Reporter

`UnirecReporter` class reports data in unirec format to NEMEA trap interface. This class needs the unirec library and is only available when the `COMPILE_WITH_UNIREC` option is enabled.

Firstly, the `UnirecReporter` obtains a reference to `UnirecOutputInterface` class from the unirec++ library in the constructor. Then, IDs of the target unirec fields must be set via the `updateUnirecFieldIDs()` method. After that, the reporter is ready to use. The `onRecordStart()` resets the current index to the unirec IDs vector, and every call of the `report()` method takes the current unirec ID from the vector, sets the according field in unirec record to the obtained value, and increments the pointer to the unirec ID vector. The `onRecordEnd()` sends the unirec record to the output interface, and the `flush()` method instructs the libunirec to send all currently pending records to the trap interface.

The possible use and strength of this reporter are demonstrated on the interconnection with Active Learning Framework in Section 5.2.

4.8 Utils

WIF also contains several utils classes needed by other parts. Currently, the utils part consists of `IpPrefix` (already described in Subsection 4.5.1), `Timer`, and `TypeTraits`.

The `Timer` class uses the standard C++ threads library `std::thread` and provides functionality to run a function periodically in another thread. The `Timer` class takes a number of seconds between each function call and a unique pointer to a child of `TimerCallback`. `TimerCallback` is an abstract class with the interface shown on listing 7. The `onStart()` method is called when the new thread is started and can be used for initialization. The `onEnd()` method is the last thing before the thread run ends and can be used for finalization. The `onTick()` method is called periodically every N seconds (based on what value was passed as the constructor's argument). This process is also shown on Figure 4.5.

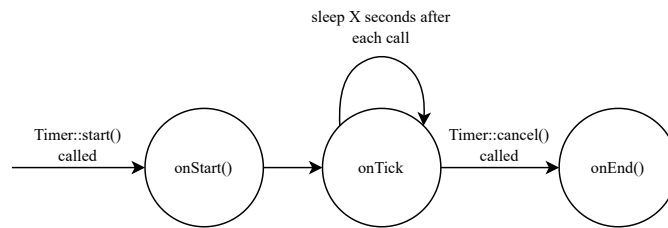


Figure 4.5: Process of TimerCallback utilization in Timer class

```

1 // Note: virtual keywords, std namespace,
2 //       and parameter names are omitted
3 //       for increased code readability
4
5 class TimerCallback {
6 public:
7     virtual void onStart() {};
8     virtual void onTick() = 0;
9     virtual void onEnd() {};
10 };
  
```

Listing 7: TimerCallback Interface

4.9 Discussion

Weak Indication Framework is a stand-alone C++ library that can be used for network traffic classification and possibly other use cases. It achieves high efficiency and can be used to develop detection modules suitable for high-speed ISP-level networks. It provides Machine Learning, pattern matching, and blacklist-based classification methods. Moreover, it provides basic data fusion algorithms and more advanced Dempster-Shafer Theory principles. Last but not least, WIF provides reporters for high explainability and active learning approaches.

The strength of WIF is that even if WIF does not contain the needed classifier, combinator, or reporter, the user can implement it on its own and use it just the same as if it was part of WIF. However, we plan to add more classification, data fusion, and reporting capabilities in the future. Unfortunately, it is out of the scope of this thesis. Nevertheless, our ideas for future WIF-related development are as follows:

MLPack MLPack is highly efficient C++ library for Machine Learning. A new classifier using MLPack can provide much faster throughput than the one based on scikit-learn.

Combinators Supply more combinators to the WIF, such as a combinator for a weighted average.

Database Reporting A reporter which sends data to a database (SQLite and others) could be used for permanent data storage and easy-to-use analysis of WIF-based detection methods.

Network Classification Prototypes

With the WIF designed and ready-to-be-used, we can now design detectors for network traffic classification and threat detection based on the newly created library. The development and functionality of WIF-based detectors are described in this chapter, emphasizing the usage of WIF.

5.1 Introduction

Several network traffic detectors will be designed and based on the Weak Indication Framework. It will allow us to verify the architecture, functionality, and the overall implementation of the WIF. The detectors described in this chapter were chosen based on the consultation with the thesis supervisors. Detectors are implemented as NEMEA modules that use the WIF library. Implemented modules are far more complex than as described in this thesis. Unfortunately, the full description is out of the scope of this thesis. We would kindly reference the reader to the Section C and the attached source codes.

5.2 Cryptomining Detector

Cryptomining Detector is a C++ implementation of the DeCrypto system [74], previously described in Section 3.6. It is the first detector based on the WIF library. The Cryptomining detector is a stream-wise NEMEA detection module that processes flows from the input and sends to its output only flows with confirmed cryptomining communication.

The repository structure of this detector is similar to the WIF library's repository. CI/CD is utilized in a similar way: code style compliance and common programming errors are checked via clang tools. Furthermore, RPM build is set up, and the RPM package can be created by calling `make rpm`.

The Cryptomining detector has one input unirec interface from which it receives incoming flow data. Similarly as in [74], the module has a prefilter in place: at least eight packets in both directions are required. Otherwise, the flow is dropped. Two interfaces are used to send data from the detector. The detector uses the primary output unirec interface to send out the miner flows. The secondary output unirec interface is used for interconnection with Active Learning Framework (ALF), described in Subsection 3.2.1.

As described in Section 3.6, DeCrypto internally uses three weak detectors and a so-called Meta classifier to process their outputs, as shown on the Figure 5.1. The first weak detector is called the Stratum detector and performs pattern-matching detection of the Stratum mining protocol. The weak detector called TLS SNI Classifier detects cryptomining-related keywords in the TLS SNI field. Lastly, ML Classifier performs statistical-based detection via Machine Learning and provides the probability of a flow being a cryptominer. Then, the Meta Classifier processes the outputs of the previously mentioned weak detectors and provides the final prediction. The weak detectors are implemented as simple wrapper classes over the classifiers and combinators from the WIF library.

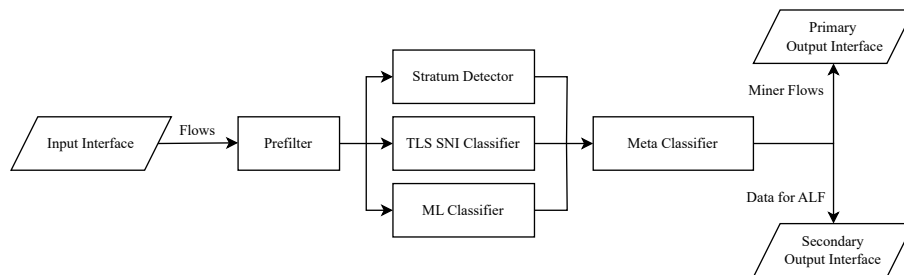


Figure 5.1: Architecture of the Cryptomining detector

Stratum detector consists of two `RegexClassifier` classes. One is used for the detection of Stratum requests, and the other one is used for the detection of Stratum responses. The regex patterns for both Stratum request and response are taken from the [78]. Moreover, both `RegexClassifier` instances use the operational mode `ANY`. Since `RegexClassifier` also requires an instance of a combinator for combining results from multiple source features, a sum combinator is used in both of them.

TLS SNI classifier also consists of two `RegexClassifier` classes. One is used for matching short cryptocurrency names (variants of `btc`, `eth`, `xmr`, and `rvn` as in [78]). The other one is used for matching suspicious mining pool keywords (`mine`, `pool`, `mining` as in [78]). Moreover, an instance of the `AverageCombinator` is used for combining the result of the two instances of the `RegexClassifier`.

ML classifier uses one of the `ScikitMLClassifier` and `AlfClassifier` classes, depending on the command line arguments. The `AlfClassifier` is used when `--use-alf` is supplied as the command line argument; otherwise, an instance of the `ScikitMLClassifier` is used. When the `ScikitMLClassifier` is used, output class probabilities are obtained from this classifier and used further in the detection process. When the ALF classifier is used, the ML features for each classified flow, the output probability array, and the timestamp of the last model load (for ML model versioning) are sent to the secondary output unirec interface via the `UnirecReporter` instance from inside the `AlfClassifier`. These data are sent to the Active Learning Framework. When the new ML model is detected on the disk, it is reloaded, and the Cryptomining detector uses the new version, keeping the model output the best as possible.

Meta classifier contains the previously described detectors, invokes them,

and performs the combination and the final detection. `BinaryDSTCombinator` class combines the TLS SNI score and the ML probability via the Dempster-Shafer Theory. Then, flows where cryptomining was detected are sent to the output unirec interface.

The Cryptomining detector requires Python libraries to be available in the system. We used the Python3.8 development package, which also contains `Python.h` header file and development package of `numpy`¹¹. These dependencies are required to successfully establish communication between the C++ detector and the `ScikitMLClassifier` via the Python C API. Furthermore, NE-MEA libraries (`libtrap`, `libunirec`, `libunirec++`) are required. The `libstdc++fs` is required for working with the filesystem. The `pthread` library is needed for the `WIF::Timer` class. Lastly, the Cryptomining detector depends, of course, on the `WIF` library.

The Cryptomining detector uses the identical detection process as the original DeCrypto system but achieves higher throughput. The performance and evaluation are described in the Chapter 6. However, it contains a new feature: the option for interconnection with ALF to keep the used ML model up-to-date. The detection process remained the same as in the original implementation.

5.3 Tor Detector

Tor Detector (TorDer) is a detector of Tor communication¹². The Tor project provides a list of Tor relay node IP addresses¹³ for others to connect. However, it can also be used to detect such communication. A stand-alone detector was developed for this purpose because we would implement the same functionality twice (once in TunDer, once in MalDer; described later) otherwise. This way TunDer and MalDer only process an existing field with the result of the Tor communication detection.

The detector's high-level architecture is shown in Figure 5.2. It processes all incoming flows, performs detection, and sends them to its output interface with new fields describing the detection result. As also depicted in the Figure 5.2, the source file with Tor relays can be updated periodically. The TorDer will automatically detect and load newer versions.

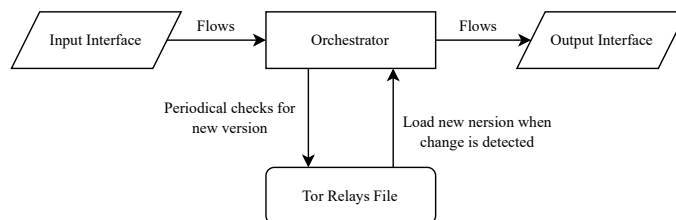


Figure 5.2: Architecture of the Tor detector

The detector has one input unirec interface for receiving incoming flow data. Each flow must contain both `SRC_IP` and `DST_IP` fields with an unirec type called `ipaddr`. The Tor detector checks if either `SRC_IP` or `DST_IP` field

¹¹<https://numpy.org>

¹²<https://www.torproject.org>

¹³<https://onionoo.torproject.org/summary?running=true>

contains a Tor relay node IP address. The WIF library performs the actual detection via an instance of the `IpPrefixClassifier` class. The detection output is true/false encoded into values one/zero and appended to the flow record. Moreover, additional information is provided, such as the direction in which the Tor relay was detected. The value `-1` indicates that the Tor relay was detected in `SRC_IP`, and the value `1` indicates that the Tor relay was detected in `DST_IP`.

Tor relay IP addresses are loaded from a file from disk. The file path is provided as a command line argument. The file must contain one IP address per line. Empty lines and lines starting with `#` (comments) are skipped. In addition, the `Timer` and `FileModificationChecker` classes from the WIF library's `utils` section are used for periodical checking for a new file version. Therefore, an additional CRON script can be set up to update the used blocklist (for example) once per day, and `TorDer` will automatically use the newest version of Tor relays.

The flows on the detector's output interface have two new `unirec` fields with the detection result: `TOR_DETECTED` and `TOR_DIRECTION`. The `TOR_DETECTED` field is of type `uint8` and holds either `0` or `1`. The `TOR_DIRECTION` is of type `int8` and holds `-1` if Tor relay was detected in `SRC_IP`, or `1` if Tor relay was detected in `DST_IP`.

The Tor detector requires the NEMEA libraries (`libtrap`, `libunirec`, and `libunirec++`). It also depends on the `stdc++fs`, `pthread`, and, of course, the WIF library. CMake is used for the build process, and RPM is used for packaging and distribution.

5.4 Tunnel Detector

Tunnel Detector (`TunDer`) is a NEMEA module designed and developed from scratch. It is not an implementation of another system that was already designed. `TunDer` aims to detect communication tunnels, such as Virtual Private Networks (VPNs), Tor, and more. The detector is distributed as the RPM package.

The Tunnel detector follows a weak-detector architecture, shown in Figure 5.3. However, `TunDer`, rather than performing the detection itself, aggregates information over time instead. Once in a while, the detector goes through the results of weak detectors for each IP address. The decision to send an alert to the output interface is made based on what combination of indicators was triggered.

A significant difference from other detectors is that `TunDer` does not perform detection for every received flow. Rather than that, the user needs to provide a configuration file with IP ranges, which the `TunDer` will observe. `TunDer` only works with flows with at least one IP address (`SRC` or `DST` IP) from one of the provided IP ranges. Otherwise, the flow is dropped and not processed.

Moreover, the Tunnel detector works in time intervals. Internal indicator stores are empty after the start of the detector. The first time window starts, and the stores are updated based on the incoming flows. The time window expires when a flow with a timestamp in the `TIME_LAST` field is received and the time difference from the window start is greater than the window size.



Figure 5.3: TunDer architecture

Then, the detector goes through the weak-detector results for each seen IP address. The results of each detector are passed into rules. If a rule is satisfied, an alert representing this rule is sent to the output interface. The default time window size is 15 minutes (900 seconds).

5.4.1 Data Stores

Weak detectors need some data storage to hold information about the detection for each IP address. The Tunnel detector uses two types of data stores for such purpose: `BinaryStore` and `CounterStore`. The binary store is a `std::vector` of `uint8_t`. Values in this store can only be either 1 or 0 (representing if an event was or was not observed in the time window). The counter store holds a `std::vector` of `uint16_t`. Its values are counters representing the number of seen events. Both store types are pre-allocated to the size supplied in the constructor's argument. When a store is cleared, all of its members are set to 0. No additional (re-)allocation is done.

We need to translate the observed IP address to an index for effective memory access. Extremely fast non-cryptographic hash algorithm `xxHash`¹⁴ is used to map IP addresses to `XXH64_hash_t` (or `IpKey`) values. The main component of the Tunnel detector, `Orchestrator`, performs this translation. Data stores then use hash maps to translate `XXH64_hash_t` to an actual `unsigned` index to a `std::vector` or another container.

5.4.2 Weak Detectors

TunDer comprises several weak detectors: detectors of default ports, detector of Tor, detector of communication with IP addresses on user-provided blocklist, and detectors processing confidence level fields of OpenVPN, WireGuard, and SSA (described later). Every detector holds information from seen flows for each IP address. Weak detectors internally use *stores* for holding the information.

There are two weak detectors based on ports. The `OpenVpnPortDetector` looks for flows with OpenVPN default port 1194. The `WireGuardPortDetector` looks for flows with port 51820, the well-known port used by Wireguard. Both detectors hold a counter for the number of seen flows with the default port seen.

¹⁴<https://xxhash.com>

If the number of seen flows with the default port exceeds the needed threshold, the result of the detection is positive. Otherwise, the result is negative.

The following three weak detectors are processing `CONF_LEVEL` fields from `ipfixprobe` plugins¹⁵. The `CONF_LEVEL` stands for confidence level and ranges from 0 – 100. It expresses confidence in the plugin about a flow’s affiliation to a group it is meant to detect. For example, the `OpenVPN` plugin appends `OVPN_CONF_LEVEL` to a flow expressing confidence that the flow is an `OpenVPN` communication. The same applies to the `WG_CONF_LEVEL` and `WireGuard` communication. Lastly, `SSA_CONF_LEVEL` detects `SYN-SYN ACK-ACK` sequences, and its values are only 0 (`SSA` not seen) or 1 (`SSA` seen).

As mentioned in the previous paragraph, the next three weak detectors are processing `OVPN_CONF_LEVEL`, `WG_CONF_LEVEL` field, and `SSA_CONF_LEVEL`. Each detector processes one of the listed fields. Similarly to port detectors, counters are used to count the seen flows. Moreover, a minimal confidence threshold can be set: the counter is not updated if the confidence level does not exceed the needed confidence threshold. At the end of the time window, the number of seen flows must be greater or equal to the defined threshold (positive result). Otherwise, the result is negative.

In addition, the detector processing `OVPN_CONF_LEVEL` also looks for the 100% confidence level. The result of this detector can be `NEGATIVE`, `POSITIVE`, or `RESULT_OVPN_100`. It is then supplied to rules as any other result and requires no special handling.

Another weak detector is the `Tor` communication detector. It processes the output fields of `TorDer` (described in Section 5.3). Again, counters are used, and the minimal number of seen flows with detected `Tor` communication can be set.

The last weak detector performs `IP` blocklist classification and is based on a user-provided blocklist. It uses `IpPrefixClassifier` class from the `WIF` library. The threshold for a number of the seen flows communicating with blocklisted `IP` addresses can be set. Therefore, users can provide their own blocklist based on the specific traffic they want to detect, and `TunDer` will perform the detection.

$$\text{CONF_LEVEL_OPVN_100} \tag{5.1}$$

$$\text{CONF_LEVEL_OVPN} \wedge \text{CONF_LEVEL_SSA} \tag{5.2}$$

$$\text{CONF_LEVEL_WG} \wedge \text{CONF_LEVEL_SSA} \tag{5.3}$$

$$\text{CONF_LEVEL_OVPN} \wedge \text{DEFAULT_PORT_OVPN} \tag{5.4}$$

$$\text{CONF_LEVEL_WG} \wedge \text{DEFAULT_PORT_WG} \tag{5.5}$$

$$\text{TOR} \tag{5.6}$$

$$\text{BLOCKLIST} \tag{5.7}$$

5.4.3 Export

When the time window ends, detection results of weak detectors are obtained and passed to the rules. Then, each satisfied rule produces an alert on the output interface with the information about the `IP` address from the observed

¹⁵<https://github.com/CESNET/ipfixprobe>

ranges, results, explanations of all weak detectors, and the string representation of the matched rule. The detectors are reset, and a new time window starts.

A system for rule matching is also implemented directly in the Tunnel detector. Abstract class `Rule` defines a common interface, shown on listing 8. The constructor (line 6) takes a string representation of the rule. The virtual method `result()` on line 8 indicates if the rule was satisfied or not. The method `registerWeakResult()` serves to supply a result of a weak detector (the first argument `DetectorID` indicates the result from which the detector is being registered). `BasicRule` extends this class and takes an ID of the detector it represents. If the result of the detector with the same ID is registered, it is saved. Otherwise, nothing is done. This rule is satisfied if the detector it represents is positive. Otherwise, it is negative. Then, `AndRule` and `OrRule` classes are provided. These classes take a vector of pointers to other rules. The `AndRule` is satisfied if all the supplied rules are also satisfied. The `OrRule` requires at least one rule to be satisfied. Moreover, these advanced logical rules can automatically build their string explanation by joining the underlying rule explanations with suitable logical characters (`&&`, `||`).

```

1 // Note: virtual keywords, std namespace,
2 //       and parameter names are omitted
3 //       for increased code readability
4 class Rule {
5 public:
6     Rule(const std::string& explanation)
7     ~Rule() = default;
8     bool result() const = 0;
9     void registerWeakResult(DetectorID, DetectionResult) = 0;
10    const std::string& explanation() const noexcept;
11 };

```

Listing 8: Rule Interface

Every rule is attempted for each IP address. The first rule (5.1) is satisfied when at least one flow with `OPVN_CONF_LEVEL` with value 100 was seen. Rules (5.2), respectively (5.3), is satisfied when both detectors of SSA and OpenVPN confidence level, respectively WireGuard confidence level, are positive. Rules (5.4), respectively (5.5) is satisfied when both detectors of default port and OpenVPN confidence level, respectively WireGuard confidence level, are positive. Rule (5.6) is satisfied when a Tor detector is positive. Finally, rule (5.7) is satisfied when the blacklist detector is positive.

5.4.4 Build Dependencies

The Tunnel detector depends on the same libraries as the Tor detector: the NEMEA libraries (`libtrap`, `libunirec`, and `libunirec++`), `stdc++fs`, `pthread`, and the `WIF` library.

5.5 Malware Detector

Malware Detector (MalDer) is a C++ implementation of the BOTA system based on the WIF library. The original BOTA system was introduced in [73] and is described in Section 3.5. Its purpose is to reveal botnet malware targeting IoT devices like Mirai, Gafgyt, and Tsunami. The detector is distributed as the RPM package.

MalDer uses a similar architecture as the Tunnel detector: flows from only supplied IP ranges are processed, multiple indicators are investigated by their corresponding weak detectors, the module as a whole works in time windows, and rules are matched for each seen IP address after the time window expires. However, the BOTA system (and MalDer) also works in 5-minute windows. The Figure 5.4 shows a high-level visualization of the Malware detector with all its weak detectors.



Figure 5.4: MalDer architecture

As mentioned earlier, the Malware detector also requires a configuration file with IP ranges to be observed. Received flows that do not belong to any of the observed IP ranges are dropped. Again, blank lines and comments (lines starting with #) are filtered out. IP ranges must be provided in CIDR format (Subnet Address/Mask Length, such as 10.0.0.0/24).

5.5.1 Data Stores

Weak detectors use stores to hold information about observed IP addresses and events. The same `BinaryStore` and `CounterStore` classes are used as in the Tunnel detector. Furthermore, two new types of stores were designed for the Malware detector.

`UniqueStore` is a templated store class for observing a number of unique values. For example, it can be used to track the number of unique destination ports (with the template argument `uint16_t`) where an IP address was connecting. Another use case is to observe with how many unique IP addresses the observed IP address was communicating. It is used in weak detectors handling anomalous traffic.

`AnomalyStore` is an advanced store utilizing Welford and Brown algorithms. It holds a cumulative sum of a metric over time. The current state of the

algorithms is updated via the new observations and can provide a prediction for the next value. This store is also used in the anomaly detectors. Its exact usage is described later.

The same process for translating IP address to an index is used as in the Tunnel detector, described in Subsection 5.4.1. Firstly, IP address is translated into an `XXH64_hash_t` (`IpKey`) value. Data stores then perform translation from an `IpKey` to an `unsigned` value used for accessing the data.

5.5.2 Weak Detectors

The Malware detector consists of the eight following weak detectors: anomaly detectors of a number of transferred bytes, packets, and the number of unique IP addresses and ports used for communication, and detectors of Command & Control communication, DHT, Stratum, and Tor protocols.

Weak detectors of DHT and Stratum protocols use the `RegexClassifier` from the WIF library. The used regex patterns are taken from the original BOTA implementation in [73]. Each of these weak detectors uses the `BinaryStore`. If the store contains the value 1, the representing event (detection of Stratum, DHT) already happened. Therefore, no additional work must be done. On the other hand, the stored value 0 means that the corresponding event has not yet occurred. Therefore, the weak detector will perform the detection. If the event was detected, the corresponding value is updated.

The weak detector of Tor communication processes the output of the Tor detector (described in Section 5.3): `TOR_DETECTED`. It does not perform any detection on its own. The binary store is utilized in the exact same way as in the weak detectors of Stratum and DHT protocols.

A weak detector called `CNCDetector` performs statistical detection of Command & Control communication, hence its name CNC. This weak detector internally uses the `ScikitMLClassifier` from the WIF library and AdaBoost ML model proposed in [73]. ML bridge must also be provided. The Malware detector uses the same Python bridge as the Cryptomining detector. Similarly as in all previously described weak detectors, binary store is utilized for tracking if C&C communication was already detected for each IP address.

Furthermore, several weak detectors perform anomaly detection. The parent abstract class `AnomalyDetector` handles the actual anomaly detection; meanwhile, its child classes are customized to observe concrete metrics. Anomaly detectors work in their own time intervals of one-minute windows. This means that the anomaly detector finished five one-minute time windows in one five-minute window of the whole BOTA system. An anomaly can occur during every one-minute window, and a maximum of 5 anomalies can be seen in one 5-minute BOTA window.

For the period of 1 minute, the value of the observed metric is updated by incoming flows (for example the number of transferred bytes from the received flow is added to the current value of all transferred bytes by this IP address). When the time window expires, anomaly detection is performed. `AnomalyStore` (with Brown and Welford algorithms) is used for getting a prediction of the observed metric. If the current value is greater than the predicted value, it is considered an anomaly, and the detector increments the number of anomalies. Then, `AnomalyStore` is updated with the current value to reflect it in the prediction the next time the window ends.

Such detection settings produced many anomalies. However, MalDer follows the proposed mechanism in BOTA: an additional criterion for an anomaly to be considered an actual anomaly. The metric value must also exceed a minimum threshold. For example, the observed value X would be considered an anomaly, but it does not exceed the minimum threshold Y ($X < Y$). Therefore, the result in that time window is not anomalous, and the number of anomalies seen is not incremented. For example, if the number of transferred bytes was 10 000 and the predicted value was 7500, it would be considered an anomaly. However, it is not greater than 1 000 000 (the default bytes anomaly threshold) and, therefore, not considered an anomaly.

When the 5-minute BOTA window expires, the result of the anomaly detector must be determined. This is done by comparing the number of anomalies seen with the number of anomalies needed to be positive (currently set to one). Then, the number of anomalies is reset, and a new time window starts.

Four weak detectors for anomaly detection are part of the Malware detector. Anomaly detection is performed in the part class `AnomalyDetector`. Its children differ in stores used and how they work with the received flow. One for observing each of the following metrics:

Bytes The sum of the number of bytes from all the received flows for a particular IP address. Performed by `BytesAnomalyDetector`.

Packets The sum of the number of packets from all the received flows for a particular IP address. Performed by `PacketsAnomalyDetector`.

IP Addresses The number of unique destination IP addresses to which the observed IP address connected. Performed by `IpAnomalyDetector`.

Ports The number of unique destination ports to which the observed IP address connected. Performed by `PortsAnomalyDetector`.

5.5.3 Export

After the 5-minute window expires, a process similar to that of the Tunnel detector takes place. Result of every weak detector is registered into the rule mechanism; see Subsection 5.4.3. An alert is created and sent to the output unirec interface for every satisfied rule. The rules are same as in the Section 3.5 and [73] and are also shown on (5.8), (5.9), and (5.10). Detectors are reset, and a new time window starts. The output alert contains the target IP address, the string representation of the matched rule, and results and explanations of each weak detector.

$$\text{C\&C} \wedge (A_b \vee A_p \vee A_{dip} \vee A_{dp}) \tag{5.8}$$

$$\text{Tor} \wedge (A_b \vee A_p \vee A_{dip} \vee A_{dp}) \tag{5.9}$$

$$\text{DHT} \wedge \text{Stratum} \tag{5.10}$$

Furthermore, the maximum number of observed IP addresses at once can be set via input argument. It means that when a new flow is received, internal data stores and translation objects can be full. Least Recently Used (LRU) cache is used to address this issue. The least recently used `IpKey` is obtained from the LRU cache and premature detection is performed. Then, this

key is removed from the internal structures for data storage and IP-to-index translation, making a free spot for a newly received flow.

5.5.4 Build Dependencies

The Malware detector depends on the NEMEA libraries (libtrap, libunirec, and libunirec++), stdc++fs, pthread, and the WIF library. In addition, Python3 libraries and numpy are required.

5.6 Discussion

The detectors described in this chapter were designed and developed later than the Weak Indication Framework, and all of them use features provided by WIF. However, several features were identified as suitable to become part of the WIF during the detectors' development.

The first identified feature suitable for becoming part of WIF are data stores. Both TunDer and MalDer use `BinaryStore` and `CounterStore` to keep information about observed IP addresses. Since detection methods can also be based on an aggregation of information, it would be beneficial for WIF to contain prepared classes for storing information with an IP address object as a key.

Rule matching is another possible improvement identified. Both TunDer and MalDer use rules representing which weak detectors have to produce a positive result for an alert to be sent to an output interface. The WIF with similar rule-matching functionality can further speed up the development of new modules and reduce the need to re-implement the code. Furthermore, rules used by both detectors are now hard-coded. Reading and parsing rules from a configuration file would make WIF-based modules more flexible and make deploying detectors to different networks easier.

Weak Indication Framework currently provides classifiers with detection methods. However, every implemented detector uses wrapper classes, which usually implement some pre-filtering, storing the data, and providing explanatory strings. Again, both TunDer and MalDer use `Orchestrator` class for invoking detectors, managing the time window, and rule matching. Moreover, abstract `WeakDetector` class serves as a general detector class with the common interface of methods such as: `accept()`, `update()`, `result()`, and `explain()`. WIF could provide a similar class to ease the development. Not only detection methods would be prepared, but also a general structure for detection modules.

Evaluation

Verifying that the implemented software in previous chapters works correctly and that the output alerts contain a valid and truthful description of a detected event is essential. Therefore, this chapter describes the evaluation process. The basic functionalities of each detector are verified, followed by performance evaluation. A description of metrics and methodology is also provided.

6.1 Metrics & Methodology

Evaluation of detection and classification processes is usually performed on labeled data. Therefore, the correctness of output labels can be verified, and statistics can be calculated. Data label and output classification label can have one of the following relationships:

True Positive (TP): positive result marked correctly as positive

False Positive (FP): negative result marked incorrectly as positive

True Negative (TN): negative result marked correctly as negative

False Negative (FN): positive result marked incorrectly as negative

Runtime is measured by standard `time` utility of UNIX systems. The flows per second (FPS) metric is then calculated as the total number of processed flows divided by the runtime in seconds. All performance and timings tests were performed on a single virtual machine with the parameters shown in Table 6.1. Detectors in the CESNET environment typically run on virtual machines with similar parameters; therefore, performing efficiency evaluation and other testing in a similar environment is ideal. We could achieve better results with the more powerful non-virtual machine. However, we want to test the detectors in the environment most likely to be used for deployment. This will give us realistic numbers, which can be expected when deployed.

Each detector was run 100 times when evaluating its performance. The GNU's `time` utility was used to measure the run time. Then, the flows per second metric were calculated as the number of flows processed by the detector divided by the runtime in seconds (*elapsed* part of the `time`'s output). Minimum, maximum, average, and median values for both metrics are provided.

Table 6.1: Parameters of the machine used for testing

Parameter	Value
System	Oracle Linux 8
Processor	Intel(R) Xeon(R) Gold 6226R
Num. of CPU cores	4
CPU frequency	2.90 GHz
L1 data cache size	32 K
L1 instruction cache size	32 K
L2 cache size	1024 K
L3 cache size	22528 K
RAM size	5.8 GB
Swap size	2 GB

Table 6.2: DeCrypto parameters used for evaluation

Parameter	Value
DST threshold	0.44
ML threshold	0.996
Buffer size	100 000
ML model	data_symmetry.pickle
Debug/verbose mode	Enabled

6.2 Cryptomining Detector

6.2.1 Correctness

Cryptomining detector is based on the DeCrypto system presented in [74]. Fortunately, [74] also presents two datasets that we can use to verify that our C++ implementation produces the same output as the original DeCrypto system. Moreover, we can compare the performance of both implementations.

Firstly, we verified that both implementations of the DeCrypto detector produce the same results. Two datasets from [78] were used: `design` with 2 094 903 flows and `evaluation` with 1 075 576 flows. The same settings were used for both implementations, shown in the Table 6.2. The datasets contain traffic captured directly on the national CESNET3 network. The `desing` dataset contains traffic from December 2021 until February 2022. The `evaluation` dataset contains traffic from February 2022 until March 2023.

We expect both implementations to produce the same output. Moreover, the original ML model from [78] was used. The Table 6.3 shows numbers of TP, FP, FN, and TN from both implementations (the original one is marked as the *Python-based* and the new is marked as the *WIF-based*). As we can see, both implementations provide the same results. The table with detailed results of weak detectors is available in Section A.1. Therefore, the newly implemented WIF-based C++ variant works correctly.

6.2.2 Throughput

Then, we decided to measure the throughput and performance of both implementations. Firstly, we created a sample from the `design` dataset on which the

Table 6.3: DeCrypto results

Dataset	Result	Python-based	WIF-based	Same?
design	TP	557 692	557 692	✓
design	FP	6	6	✓
design	FN	25 864	25 864	✓
design	TN	966 156	966 156	✓
evaluation	TP	278 964	278 964	✓
evaluation	FP	5	5	✓
evaluation	FN	58 827	58 827	✓
evaluation	TN	489 115	489 115	✓

tests will be performed. This sample contains 250 000 miner flows and 250 000 of non-miner flows in a random order. Therefore, the dataset is balanced, and the order of flows is random, simulating real traffic coming into the detector. Then, each detector was run 100 times, and its runtime was measured via the GNU's `time` utility. The statistics are calculated from the `elapsed` output of the command showing the total time needed for a program to run.

The Table 6.4 provides the overview of the results. Both runtime in seconds and flows per second statistics are provided. We can see the dramatic and significant improvement: the newly implemented C++ version is 7.5 times more effective than the original Python implementation. Therefore, it has sufficient throughput to operate on national high-speed networks where thousands of flows per second are sent to the detector.

Table 6.4: Performance of DeCrypto systems

Seconds				
Variant	Min [s]	Average [s]	Median [s]	Max [s]
Python-based	30.77	32.25	31.98	34.94
WIF-based	4.42	4.58	4.55	5.05

Flows per second				
Variant	Min [fps]	Average [fps]	Median [fps]	Max [fps]
Python-based	14 310.25	15 502.96	15 637.22	16 249.59
WIF-based	99 009.9	109 070.28	109 890.11	113 122.17

6.3 TorDer

6.3.1 Correctness

TorDer is a simple module that performs blocklist-based detection. Therefore, we will first assess if the detector works correctly. A simple blocklist and a special trapcap file for the NEMEA system will be created. TorDer will be run with the generated files, and its output will be evaluated.

A blocklist was generated containing 1000 IPv4 and 1000 IPv6 randomly generated addresses. Moreover, a flow data file was created with the following:

5000 IPv4 and 5000 IPv6 addresses from the blocklist together with 10 000 IPv4 and 10 000 IPv6 addresses not on the blocklist for the in the SRC_IP field. An IP address is present in the DST_IP field which is not on the blocklist. We expect that 10 000 flows will be marked as Tor communication and 20 000 will be marked as non-Tor. TorDer ran with the following data, and blocklist produced the expected 10 000 Tor flows and 20 000 non-Tor flows. Therefore, the expected output was obtained, and we can say that the Tor detector works correctly.

6.3.2 Thoughtput

Then, we decided to evaluate the performance of the Tor detector. The same script for blocklist and data generation was used to create a blocklist with 50 000 IPv4 and 50 000 IPv6 addresses and test data with 500 000 flows on the blocklist and 500 000 flows not on the blocklist for each IP version, totaling to 1 000 000 blocklisted flows and 1 000 000 non-blocklisted flows. Again, we run the detector 100 times with the generated test data and blocklist. The statistics of measured runtimes are provided in the Table 6.5. As we can see, TorDer achieves very high throughput and can serve even very large networks where tens of thousands of flows are sent to the detector each second.

Table 6.5: Performance of Tor detector

Seconds			
Min [s]	Average [s]	Median [s]	Max [s]
3.01	3.12	3.1	5.69

Flows per second			
Min [fps]	Average [fps]	Median [fps]	Max [fps]
395 430.58	723 882.8	725 806.45	747 508.31

6.4 TunDer

6.4.1 Correctness

A file with specially crafted flow data was created for each rule internally matched by TunDer. Each file contains three records: the initial flow with all the fields zeroed, the second field specially crafted to match the targeted rule, and the third flow outside the aggregation window so TunDer will perform rule matching. Adequate blocklist and IP range definition files were created as well. If TunDer works correctly, each output file should contain exactly one alert representing the matched rule it was specially crafted for. Moreover, a file that should trigger no rule (R8_NO_RULE) was crafted to verify that no unexpected alerts are sent to the output. The Table 6.6 shows test results. We can see that all tests have been passed, and TunDer's rule matching works as expected.

The Tunnel detector itself does not implement any advanced or complex detection mechanism; instead, it aggregates information together during a specific time period. Blocklist-based detection and detection of defined port numbers

Table 6.6: TunDer rule tests

File	Targeted Rule	Passed?
R1_OPVN_100	OVPN_CONF_LEVEL_100_SEEN	✓
R2_OPVN_AND_SSA_CONF	OVPN_AND_SSA_CONF	✓
R3_WG_AND_SSA_CONF	WG_AND_SSA_CONF	✓
R4_TOR	TOR	✓
R5_BLOCKLIST	BLOCKLIST	✓
R6_OVPN_CONF_AND_PORT	OVPN_CONF_AND_PORT	✓
R7_WG_AND_PORT	WG_CONF_AND_PORT	✓
R8_NO_RULE	None	✓

take place, but both methods are elementary methods for network classification, and the previous tests of rules proved their correctness. Furthermore, the detection of OpenVPN and WireGuard protocols and SSA sequence is performed directly in the flow exporter, and TunDer only works with the finished probabilities. Therefore, we cannot test these methods since they are not directly part of our detector.

6.4.2 Throughput

The performance of the Tunnel detector was evaluated by running the detector 100 times on data captured directly on the CESNET3 network. A filter for capture was used so that only flows from the CESNET IP ranges (later used in the detector) were captured with at least three packets in both directions. This will ensure that the data used for performance testing will be processed by TunDer and not dropped (TunDer’s prefilter drop flows, which do not contain any IP address from defined protected ranges, see Section 5.4). Before any performance testing, we put a debug print of the number of processed flows to ensure that all 2 225 000 captured flows were processed. After we confirmed that all the flows would be processed, the debug print was removed. The Table 6.7 shows the measured times and flows per second. We can see that even though the Tunnel detector is more complex than the TorDer and contains more detectors, it still achieves high throughput and can be deployed to large-scale national-level networks.

Table 6.7: Performance of TunDer detector

Seconds			
Min [s]	Average [s]	Median [s]	Max [s]
3.21	3.34	3.31	3.51

Flows per second			
Min [fps]	Average [fps]	Median [fps]	Max [fps]
641 025.64	673 755.35	679 758.31	700 934.58

6.5 MalDer

6.5.1 Correctness

Firstly, the basic functionality of the Malware detector will be tested. Specially crafted flow data will be used to verify that each rule works correctly and is matched when expected. MalDer uses three rules, but two of them contain the following part: $(A_b \vee A_p \vee A_{dip} \vee A_{dp})$. We created four versions of one rule, which uses this clausula for each anomaly detector to verify that all possibilities work (rules R2a—R2d in the Table 6.8). Moreover, a file that should trigger no rule (R4_NO_RULE) was crafted to verify that no unexpected alerts are sent to the output. The Table 6.8 shows that all tests were passed, meaning that every weak detector in MalDer and the rule matching works correctly and as expected.

Table 6.8: MalDer rule tests

File	Targeted Rule	Passed?
R1_DHT_AND_STRATUM	DHT_AND_STRATUM	✓
R2a_TOR_AND_ANOMALY_BYTES	TOR_AND_ANOMALY_BYTES	✓
R2b_TOR_AND_ANOMALY_PACKETS	TOR_AND_ANOMALY_PACKETS	✓
R2c_TOR_AND_ANOMALY_IP	TOR_AND_ANOMALY_IP	✓
R2d_TOR_AND_ANOMALY_PORT	TOR_AND_ANOMALY_PORT	✓
R3_CNC_AND_ANOMALY	CNC_AND_ANOMALY	✓
R4_NO_RULE	None	✓

6.5.2 Throughput

The performance of the Malware detector was evaluated on the same data captured for evaluating TunDer; see Section 6.4 for more information. The test data contains only flows from protected ranges, meaning that MalDer will drop no flow, and all flows will be processed. Therefore, the number will show the actual number of flows the detector can process per second. The Table 6.9 shows the measured times and flows per second. The Malware detector has a significantly lower throughput than the TorDer and TunDer. However, it is far more complex than any of the other detectors. Still, when a pre-filter filters out short flows representing scans and similar short flows holding no information, it can be used on large-scale networks. Moreover, we can still consider the measured throughput a success because the original BOTA implementation was not deployed on the CESNET3 network due to performance issues.

6.6 Discussion

Each detector based on the WIF was thoroughly tested: basic functionality tests performed on specially crafted data, and performance tests showing the theoretical number of flows that detectors can process. The tests showed that the Weak Indication Framework can be used to develop highly efficient detectors capable of operating on high-speed national-level networks and that the produced output alerts provide reliable and meaningful output.

Table 6.9: Performance of MalDer detector

Seconds			
Min [s]	Average [s]	Median [s]	Max [s]
41.15	43.79	43.73	50.3

Flows per second			
Min [fps]	Average [fps]	Median [fps]	Max [fps]
44 731.61	51 457.66	51 452.1	54 678.01

In addition, the efficiency evaluation was performed in an environment very close to the real-world deployment. Therefore, it is reasonable to expect similar flows per second performance as measured. The data used for testing was also captured on the national network CESNET3 and resembled real-world traffic.

Deployment

7.1 Targeted Network

The developed detectors target the CESNET3 network. The CESNET3 network is a national research and educational network operated by CESNET, the Czech National research and education network (NREN) operator. The network is depicted on the Figure 7.1. We can see that the CESNET3 is a high-speed network with backbone lines reaching speeds up to 400 Gbps. Moreover, the network has more than half a million daily users.

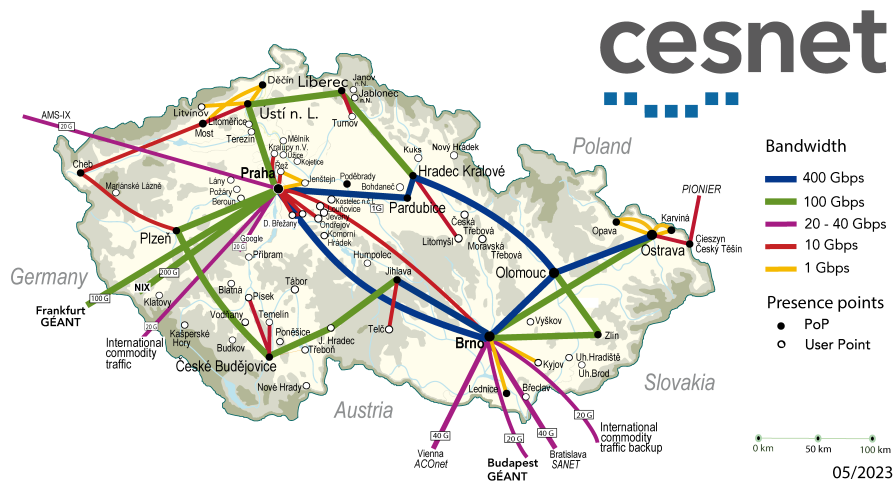


Figure 7.1: National network CESNET3

Network probes with the ipfixprobe software are used to monitor the CESNET3 network. Captured network flow telemetry is then sent to the central collector server as messages in IPFIX format. The central collector serves as a proxy. Flow data are distributed from the proxy collector to other servers for processing.

The processing servers use the NEMEA system for stream-wise network traffic classification and threat detection. All detectors implemented in this thesis were deployed to the same virtual server in the CESNET infrastructure.

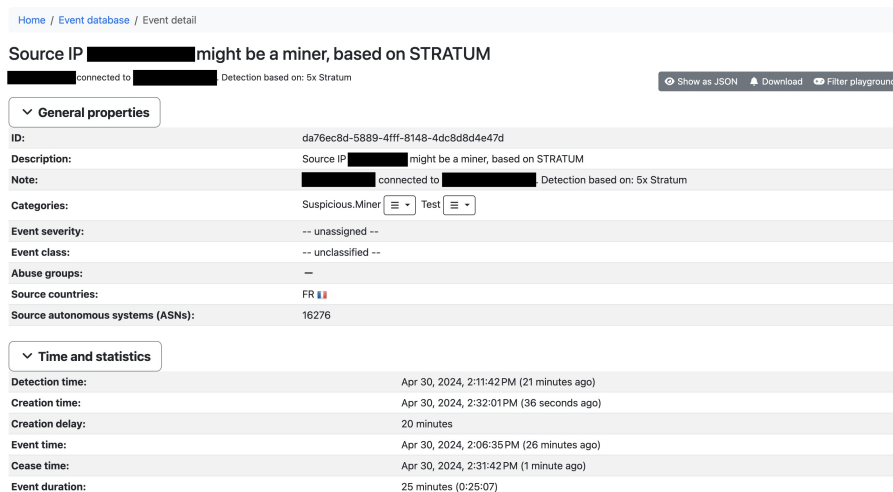
The server receives the flow data from the proxy collector from the CESNET3 network and processes it in real time.

Alerts produced by threat detectors are sent to a central database system called Warden. Network security operators from the CSIRT team then access the stored alerts when an incident occurs. Mentat is another system developed by CESNET that is used to browse and access the alert database.

7.2 Cryptomining Detector

The Cryptomining detector was deployed to the CESNET3 network several months ago. Since then, alerts from this detector have been sent to the Warden system in CESNET. The detector produces around 400 alerts per hour (see the Section B.1 in the Appendix). We consider this number to be reasonable when we take into account the size of the network. The alert of the Cryptomining detector is shown on the Figure 7.2, as it can be seen in the Mentat system.

The same settings were used as when the original DeCrypto implementation was deployed in [78]. Therefore, we did not have to experiment with the detector's settings. Moreover, we did not encounter any problems while deploying the Cryptomining detector.



Home / Event database / Event detail

Source IP [redacted] might be a miner, based on STRATUM

[redacted] connected to [redacted] Detection based on: 5x Stratum

Show as JSON Download Filter playground

General properties

ID: da76ec8d-5889-4fff-8148-4dc8d8d4e47d

Description: Source IP [redacted] might be a miner, based on STRATUM

Note: [redacted] connected to [redacted] Detection based on: 5x Stratum

Categories: Suspicious.Miner Test

Event severity: -- unassigned --

Event class: -- unclassified --

Abuse groups: --

Source countries: FR

Source autonomous systems (ASNs): 16276

Time and statistics

Detection time: Apr 30, 2024, 2:11:42 PM (21 minutes ago)

Creation time: Apr 30, 2024, 2:32:01 PM (36 seconds ago)

Creation delay: 20 minutes

Event time: Apr 30, 2024, 2:06:35 PM (26 minutes ago)

Cease time: Apr 30, 2024, 2:31:42 PM (1 minute ago)

Event duration: 25 minutes (0:25:07)

Figure 7.2: Screenshot of alert in the Mentat web interface

7.3 Tor Detector

The Tor Detector was deployed to the CESNET3 network during the winter of 2023. However, its output is not sent to the SIEM systems. Tunnel and Malware detectors are connected to the TorDer and further process the flows from its output. The Tor detector is a simple module with no additional parameters, so no setting adjustments had to be made. The only other task was to set up a CRON job to update the Tor relay spreadsheet file, which we successfully did.

7.4 Tunnel Detector

The Tunnel detector was deployed to the CESNET3 around the same time as the Tor detector. Its output was thoroughly examined during the months, and multiple settings were adjusted. Moreover, we experimented with different combinations of rules. For example, the result of our experiments is the rule satisfied when OpenVPN confidence level equal to 100 was seen (described in the Section 5.4). The minimal thresholds for anomaly detectors were also adjusted multiple times during the deployment.

Furthermore, I connected via the WireGuard VPN to the CESNET infrastructure while the TunDer was running, and my public IP address produced an alert that detected WireGuard usage. The thesis supervisor, Ing. Karel Hynek, Ph.D., performed the same test with the same result. We consider this a success story, proving that the detector works correctly and can provide meaningful and truthful output even on large networks.

7.5 Malware Detector

The Malware detector was the last detector deployed to the CESNET3 network. Thanks to its deployment on live traffic, several segmentation faults were discovered and fixed.

At first, the detector produced too many alerts, mainly due to anomaly detection. Thresholds with minimal anomaly values were adjusted to mitigate this problem. Such measures lowered the number of alerts produced to a feasible number. Moreover, alerts describing anomalous traffic became much more meaningful because alerts with low values were no longer produced. Furthermore, the BOTA instance was run on the same live data as the Malware detector. The outputs of both detectors were identical, so we concluded that the Malware detector works correctly.

7.6 Summary

After all the detectors were deployed, our virtual collector with the NEMEA system looked like the Figure 7.3. Thanks to the deployment notes, several errors were fixed. The detectors would not have been able to run 24/7 without the opportunity to deploy them directly to CESNET's monitoring infrastructure.

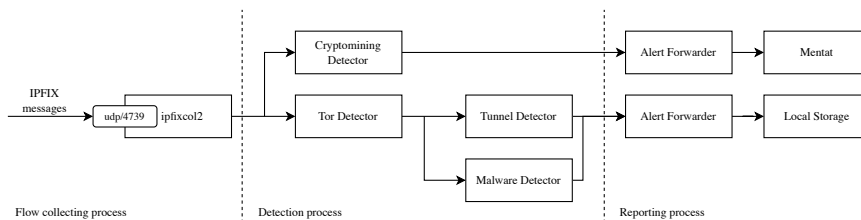


Figure 7.3: Server with deployed detectors

Conclusion

The main goal of this thesis was to research the state-of-the-art methods for network traffic classification and threat detection based on multiple classification indicators and meta classifiers. Based on the collected knowledge, this thesis describes a design of a novel library called Weak Indication Framework (WIF) containing the chosen methods useful for further development of reliable classifiers and detectors. This new technology has been developed and evaluated using several different use cases during this thesis. As a result, the following detectors have been developed and deployed in a production environment: Cryptomining detector, TorDer, TunDer, and MalDer. Each of these detectors is efficient enough to operate on the national network CESNET3 with half a million users. Moreover, the designed detectors are not only proof-of-concept as initially expected, but every one of them is a production-ready detector capable of reporting meaningful alerts even on large-scale networks.

Even though WIF met the criteria set by the thesis assignment, we identified possible features and enhancements that would further increase WIF's ability to accelerate the development process of detection modules. Our future plans are to implement features for storing and aggregating information over time, such as storage stores of binary information (an event (not)seen), counters (number of bytes/packets transferred over time), and more. Moreover, information is typically aggregated over an IP address. Therefore, utils for mapping an IP address to a store key would be useful as well. The described possible improvements were not part of the initial requirements and are out-of-scope of this thesis. However, we plan to add them in the near future.

To sum up, this thesis thoroughly surveys detection and classification methods for network traffic monitoring, an efficient C++ WIF library is proposed, and several detectors capable of operation on national-level networks with meaningful output usable by network security operators via the SIEM system were developed. Moreover, the developed software is ready to be used and deployed by others to monitor ISP, corporate, or household-level networks.

Bibliography

- [1] J. Day and H. Zimmermann, "The osi reference model," *Proceedings of the IEEE*, vol. 71, no. 12, pp. 1334–1340, 1983.
- [2] M. M. Alani, *Guide to OSI and TCP/IP Models*, ser. SpringerBriefs in Computer Science. Cham: Springer International Publishing, 2014. [Online]. Available: <https://link.springer.com/10.1007/978-3-319-05152-9>
- [3] D. Clark, "The design philosophy of the darpa internet protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 18, no. 4, p. 106–114, aug 1988. [Online]. Available: <https://doi.org/10.1145/52325.52336>
- [4] B. Leiner, R. Cole, J. Postel, and D. Mills, "The darpa internet protocol suite," *IEEE Communications Magazine*, vol. 23, no. 3, pp. 29–34, 1985.
- [5] C. M. Kozierek, *The TCP/IP guide: a comprehensive, illustrated Internet protocols reference*. No Starch Press, 2005.
- [6] S. L. Levin and S. Schmidt, "Ipv4 to ipv6: Challenges, solutions, and lessons," *Telecommunications Policy*, vol. 38, no. 11, pp. 1059–1068, 2014. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0308596114001128>
- [7] Information Sciences Institute, University of Southern California, "Internet Protocol," RFC 791, Sep. 1981. [Online]. Available: <https://www.rfc-editor.org/info/rfc791>
- [8] B. Hinden and D. S. E. Deering, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, Dec. 1998. [Online]. Available: <https://www.rfc-editor.org/info/rfc2460>
- [9] Information Sciences Institute University of Southern California, "Transmission Control Protocol," RFC 793, Sep. 1981. [Online]. Available: <https://www.rfc-editor.org/info/rfc793>
- [10] V. Cerf and R. Kahn, "A protocol for packet network intercommunication," *IEEE Transactions on Communications*, vol. 22, no. 5, pp. 637–648, 1974.

BIBLIOGRAPHY

- [11] W. Eddy, “Transmission Control Protocol (TCP),” RFC 9293, Aug. 2022. [Online]. Available: <https://www.rfc-editor.org/info/rfc9293>
- [12] J. Postel, “User Datagram Protocol,” RFC 768, Aug. 1980. [Online]. Available: <https://www.rfc-editor.org/info/rfc768>
- [13] C. Allen and T. Dierks, “The TLS Protocol Version 1.0,” RFC 2246, Jan. 1999. [Online]. Available: <https://www.rfc-editor.org/info/rfc2246>
- [14] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” RFC 8446, Aug. 2018. [Online]. Available: <https://www.rfc-editor.org/info/rfc8446>
- [15] P. Mockapetris, “Domain names - concepts and facilities,” RFC 1034, Nov. 1987. [Online]. Available: <https://www.rfc-editor.org/info/rfc1034>
- [16] P. Mockapetris, “Domain names - implementation and specification,” RFC 1035, Nov. 1987. [Online]. Available: <https://www.rfc-editor.org/info/rfc1035>
- [17] OpenVPN, “OpenVPN,” visited on 2024-02-15. [Online]. Available: <https://openvpn.net>
- [18] WireGuard, “WireGuard,” visited on 2024-02-15. [Online]. Available: <https://www.wireguard.com>
- [19] R. T. El-Maghraby, N. M. Abd Elazim, and A. M. Bahaa-Eldin, “A survey on deep packet inspection,” in *2017 12th International Conference on Computer Engineering and Systems (ICCES)*, 2017, pp. 188–197.
- [20] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee, “Internet traffic classification demystified: myths, caveats, and the best practices,” in *Proceedings of the 2008 ACM CoNEXT Conference*, ser. CoNEXT '08. New York, NY, USA: Association for Computing Machinery, 2008. [Online]. Available: <https://doi.org/10.1145/1544012.1544023>
- [21] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, “Deep packet inspection as a service,” in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 271–282. [Online]. Available: <https://doi.org/10.1145/2674005.2674984>
- [22] R. Hofstede, P. Čeleda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, “Flow monitoring explained: From packet capture to data analysis with netflow and ipfix,” *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014.
- [23] B. Claise, B. Trammell, and P. Aitken, “Specification of the ip flow information export (ipfix) protocol for the exchange of flow information,” 2013, visited on 2024-01-16. [Online]. Available: <https://www.ietf.org/rfc/rfc7011.txt>

-
- [24] B. Trammell and E. Boschi, “An introduction to ip flow information export (ipfix),” *IEEE Communications Magazine*, vol. 49, no. 4, pp. 89–95, 2011.
- [25] T. Fioreze, M. O. Wolbers, R. van de Meent, and A. Pras, “Finding elephant flows for optical networks,” in *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, 2007, pp. 627–640.
- [26] P. Aitken, B. Claise, and B. Trammell, “Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information,” RFC 7011, Sep. 2013. [Online]. Available: <https://www.rfc-editor.org/info/rfc7011>
- [27] E. W. Bethel, S. Campbell, E. Dart, K. Stockinger, and K. Wu, “Accelerating network traffic analytics using query-driven visualization,” in *2006 IEEE Symposium On Visual Analytics Science And Technology*. IEEE, 2006, pp. 115–122.
- [28] R. Soltani, D. Goeckel, D. Towsley, and A. Houmansadr, “Towards provably invisible network flow fingerprints,” in *2017 51st Asilomar Conference on Signals, Systems, and Computers*, 2017, pp. 258–262.
- [29] C. Gokhale and O. O. Olugbara, “Dark web traffic analysis of cybersecurity threats through south african internet protocol address space,” *SN Computer Science*, vol. 1, pp. 1–20, 2020.
- [30] S. Kumar, “Survey of current network intrusion detection techniques,” *Washington Univ. in St. Louis*, pp. 1–18, 2007.
- [31] H. Mokalled, R. Catelli, V. Casola, D. Debertol, E. Meda, and R. Zunino, “The applicability of a siem solution: Requirements and evaluation,” in *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2019, pp. 132–137.
- [32] T. Cejka, V. Bartos, M. Svepes, Z. Rosa, and H. Kubatova, “Nemea: A framework for network traffic analysis,” in *2016 12th International Conference on Network and Service Management (CNSM)*, 2016, pp. 195–201.
- [33] G. Navarro, “Pattern matching,” *Journal of Applied Statistics*, vol. 31, no. 8, pp. 925–949, 2004. [Online]. Available: <https://doi.org/10.1080/0266476042000270527>
- [34] R. Antonello, S. Fernandes, D. Sadok, J. Kelner, and G. Szabó, “Design and optimizations for efficient regular expression matching in dpi systems,” *Computer Communications*, vol. 61, pp. 103–120, 2015. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S014036641500002X>
- [35] Q. Hu, S.-Y. Yu, and M. R. Asghar, “Analysing performance issues of open-source intrusion detection systems in high-speed networks,” *Journal of Information Security and Applications*, vol. 51, p. 102426, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2214212619306003>

- [36] V. Gupta, M. Singh, and V. K. Bhalla, "Pattern matching algorithms for intrusion detection and prevention system: A comparative analysis," in *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2014, pp. 50–54.
- [37] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.
- [38] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977. [Online]. Available: <https://doi.org/10.1137/0206024>
- [39] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, p. 762–772, oct 1977. [Online]. Available: <https://doi.org/10.1145/359842.359859>
- [40] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.
- [41] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015. [Online]. Available: <https://www.science.org/doi/abs/10.1126/science.aaa8415>
- [42] B. Mahesh, "Machine learning algorithms-a review," *International Journal of Science and Research (IJSR).[Internet]*, vol. 9, no. 1, pp. 381–386, 2020.
- [43] R. Awati, "Garbage in, garbage out (gigo)," Jun. 2023, visited on 2024-03-04. [Online]. Available: www.techtarget.com/searchsoftwarequality/definition/garbage-in-garbage-out
- [44] T. Times, "Work with new electronic 'brains' opens field for army math experts," Nov. 1957, visited on 2024-03-04. [Online]. Available: https://www.newspapers.com/image/55787725/?clipping_id=50687334
- [45] S. B. Kotsiantis, I. Zaharakis, P. Pintelas *et al.*, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, vol. 160, no. 1, pp. 3–24, 2007.
- [46] S. Naeem, A. Ali, S. Anam, and M. M. Ahmed, "An unsupervised machine learning algorithms: Comprehensive review," *Int. J. Comput. Digit. Syst*, 2023.
- [47] J. Brabec, T. Komárek, V. Franc, and L. Machlica, "On model evaluation under non-constant class imbalance," in *Computational Science–ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part IV 20*. Springer, 2020, pp. 74–87.
- [48] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," *IEEE transactions on knowledge and data engineering*, vol. 31, no. 12, pp. 2346–2363, 2018.
- [49] B. Settles, "Active learning literature survey," *University of Wisconsin-Madison Department of Computer Sciences*, 2009.

-
- [50] J. Pešek, D. Soukup, and T. Čejka, “Active learning framework for long-term network traffic classification,” in *2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*, 2023, pp. 0893–0899.
- [51] J. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.
- [52] J. R. Quinlan, “Learning efficient classification procedures and their application to chess end games,” in *Machine learning*. Elsevier, 1983, pp. 463–482.
- [53] T. K. Ho, “Random decision forests,” in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, 1995, pp. 278–282 vol.1.
- [54] G. Biau and E. Scornet, “A random forest guided tour,” *Test*, vol. 25, pp. 197–227, 2016.
- [55] C. Stanfill and D. Waltz, “Toward memory-based reasoning,” *Commun. ACM*, vol. 29, no. 12, p. 1213–1228, dec 1986. [Online]. Available: <https://doi.org/10.1145/7902.7906>
- [56] E. Fix and J. L. Hodges, “Discriminatory analysis. nonparametric discrimination: Consistency properties,” *International Statistical Review/Revue Internationale de Statistique*, vol. 57, no. 3, pp. 238–247, 1989.
- [57] Y. Freund and R. E. Schapire, “A decision-theoretic generalization of online learning and an application to boosting,” in *European conference on computational learning theory*. Springer, 1995, pp. 23–37.
- [58] R. E. Schapire, “Explaining adaboost,” in *Empirical Inference: Festschrift in Honor of Vladimir N. Vapnik*. Springer, 2013, pp. 37–52.
- [59] S. Lloyd, “Least squares quantization in pcm,” *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.
- [60] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.
- [61] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural Networks*, vol. 61, pp. 85–117, 2015. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0893608014002135>
- [62] D. M. Hawkins, *Identification of outliers*. Springer, 1980, vol. 11.
- [63] M. Ahmed, A. Naser Mahmood, and J. Hu, “A survey of network anomaly detection techniques,” *Journal of Network and Computer Applications*, vol. 60, pp. 19–31, 2016. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1084804515002891>
- [64] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM Comput. Surv.*, vol. 41, no. 3, jul 2009. [Online]. Available: <https://doi.org/10.1145/1541880.1541882>

- [65] E. S. Gardner, “Exponential smoothing: The state of the art—part ii,” *International Journal of Forecasting*, vol. 22, no. 4, pp. 637–666, 2006. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0169207006000392>
- [66] R. G. Brown and A. D. Little, “Exponential smoothing for prediction demand,” 1956, visited on 2024-02-21. [Online]. Available: <https://www.industrydocuments.ucsf.edu/tobacco/docs/#id=jzlc0130>
- [67] F. Castanedo *et al.*, “A review of data fusion techniques,” *The scientific world journal*, vol. 2013, 2013.
- [68] Y. Bar-Shalom, F. Daum, and J. Huang, “The probabilistic data association filter,” *IEEE Control Systems Magazine*, vol. 29, no. 6, pp. 82–100, 2009.
- [69] R. E. Kalman, “A New Approach to Linear Filtering and Prediction Problems,” *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, 03 1960. [Online]. Available: <https://doi.org/10.1115/1.3662552>
- [70] G. Bishop, G. Welch *et al.*, “An introduction to the kalman filter,” *Proc of SIGGRAPH, Course*, vol. 8, no. 27599-23175, p. 41, 2001.
- [71] A. P. Dempster, “Upper and Lower Probabilities Induced by a Multivalued Mapping,” *The Annals of Mathematical Statistics*, vol. 38, no. 2, pp. 325 – 339, 1967. [Online]. Available: <https://doi.org/10.1214/aoms/1177698950>
- [72] G. Shafer, *A Mathematical Theory of Evidence*. Princeton University Press, 2021. [Online]. Available: <https://doi.org/10.1515/9780691214696>
- [73] D. Uhříček, K. Hynek, T. Čejka, and D. Kolář, “BotA: Explainable iot malware detection in large networks,” *IEEE Internet of Things Journal*, vol. 10, no. 10, pp. 8416–8431, 2023.
- [74] R. Plný, K. Hynek, and T. Čejka, “Decrypto: Finding cryptocurrency miners on isp networks,” in *Nordic Conference on Secure IT Systems*. Springer, 2022, pp. 139–158.
- [75] M. Čtrnáctý, *Softwarový modul pro rozpoznání VPN v síťovém provozu*. České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.
- [76] J. Jiráček, *Detekce VPN provozu pomocí automatu*. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.
- [77] P. Valach, *Analysis and detection of WireGuard traffic*. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.
- [78] P. Richard, “Crypto-currency miner detection from extended ip flow data,” B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2022.

Evaluation

This chapter contains detailed results from the evaluation of the Cryptomining detector and the comparison with the previous Python implementation.

A.1 Comparison of DeCrypto implementations

This section contains detailed results of both C++ and Python-based implementations of the DeCrypto detector on both the Design dataset, available in the Table A.1, and the Evaluation dataset, available in the Table A.2.

Table A.1: Comparison of DeCrypto implementations on the Design dataset

Detector	Result	Python	WIF	Same?
Stratum	TP	466558	466558	✓
Stratum	FP	0	0	✓
Stratum	FN	0	0	✓
Stratum	TN	0	0	✓
DST	TP	1363	1363	✓
DST	FP	0	0	✓
DST	FN	0	0	✓
DST	TN	401070	401070	✓
ML	TP	89771	89771	✓
ML	FP	6	6	✓
ML	FN	25864	25864	✓
ML	TN	565086	565086	✓
Meta	TP	557692	557692	✓
Meta	FP	6	6	✓
Meta	FN	25864	25864	✓
Meta	TN	966156	966156	✓

Table A.2: Comparison of DeCrypto implementations on the Evaluation dataset

Detector	Result	Python	WIF	Same?
Stratum	TP	262197	262197	✓
Stratum	FP	0	0	✓
Stratum	FN	0	0	✓
Stratum	TN	0	0	✓
DST	TP	1456	1456	✓
DST	FP	0	0	✓
DST	FN	0	0	✓
DST	TN	254027	254027	✓
ML	TP	15311	15311	✓
ML	FP	5	5	✓
ML	FN	58827	58827	✓
ML	TN	235088	235088	✓
Meta	TP	278964	278964	✓
Meta	FP	5	5	✓
Meta	FN	58827	58827	✓
Meta	TN	489115	489115	✓

Deployment Notes

This chapter contains additional information about the produced alerts by the Cryptomining detector when deployed to the CESNET3 network.

B.1 Alerts from Cryptomining Detector

The Figure B.1 shows the number of produced alerts per hour when the Cryptomining detector was deployed on the national CESNET3 network operated by Czech NREN called CESNET.

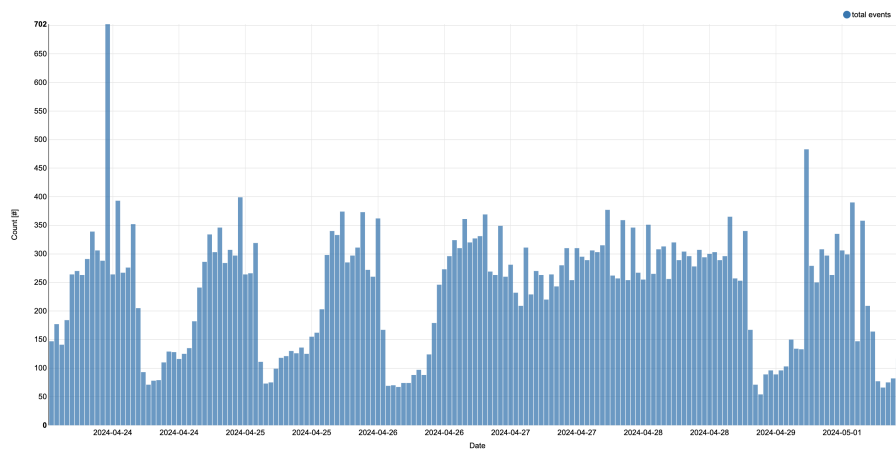


Figure B.1: Number of alerts from the deployed Cryptomining detector

User Manual

This chapter contains help sections from the developed network traffic classification prototypes.

C.1 Cryptomining Detector

DeCrypto - v2.1.2

Help

=====

DeCrypto has one input interface with incoming flow data. The primary output interface contains standard alerts. The secondary output interface contains ML features and flow info - if and only if the `--use-alf` is specified. Otherwise this interface should be set to `'b:'`.

Example without ALF: `decrypto -i u:flowData,u:alerts,b:`

Example with ALF: `decrypto -i u:flowData,u:alerts,u:alfData`

`-b`

Python Bridge Path [str]

Default: `/opt/decrypto/runtime/bridge.py`

`-d`

Enable debug mode

Default: `false`

`-f`

Flow Buffer Size [unsigned]

Default: `50000`

`-h`

Display help section [-]

Default: `false`

`-m`

ML Model Path, pickle format[str]

Default: `/opt/decrypto/runtime/rf.pickle`

```
--dst
  DST Threshold [0..1]
  Default: 0.03
--ml
  ML Threshold [0..1]
  Default: 0.99
--no-rst-fin
  Filter flows with empty SNI and RST/FIN flags
  Default: false
--use-alf
  Send ML features and flow info for ALF to the
  secondary output interface
  Default: false
```


C.2 Tor Detector

TorDer - v1.1.2

Help

=====

TorDer has one input interface with incoming flow data. The primary output interface contains copy of the input flow with new fields describing detection result. TOR_DETECTED contains 1, if either SRC_IP or DST_IP was found on current Tor relays file blocklist, 0 otherwise. Moreover, TOR_DIRECTION describes the direction: value 1 is present, if DST_IP is Tor relay, -1 is SRC_IP is Tor relay. Value 0 is set, when Tor was not detected.

--tick-interval

Interval in seconds, in which Tor relays file is checked for changes [unsigned]

Default: 15

--tor-relays-file

Tor relays file path (formatted as one IP per line) [str]

Default: None

Args must be always passed separated by space:

OK: --tick-interval 10

FAIL: --tick-interval10

C.3 Tunnel Detector

TunDer - v1.2.1

Help

=====

TunDer is a detector of covert communication tunnels.

It uses OVPN,WG,SSA_CONF_LEVEL fields for detection of OpenVPN and WireGuard. It consists of multiple weak detectors: CONF_LEVEL Detector for both OVPN, WG and SSA, Default Port Detector for both OVPN and WG, Tor Detector, and Blocklist Detector. Every detector can be customized: threshold for number of positive flows, which has to be seen in the time window, to consider detector in this time window to be positive. Moreover, a probability threshold can be set for CONF_LEVEL detectors, to define needed minimal value of CONF_LEVEL field, to consider flow positive. Results of weak detectors are observed for each IP address defined as observed. When time interval expires, rule matching takes place and every satisfied rule for each observed IP address generates an alert on the output interface, which describes results and explanations for each weak detector.

-d

Enable debug mode [-]
Default: false

-h

Display help section [-]
Default: false

-p

Enable pre-export [-]
Default: false

--flow-store-size

Flow Store Size [unsigned]
Default: 1000000

--time-window-size

Time Window Size of TunDer in seconds [unsigned]
Default: 900

--ovpn-port-threshold

Threshold for OVPN Port Detector [unsigned]
Default: 5

--wg-port-threshold

Threshold for WireGuard Port Detector [unsigned]
Default: 5

--ovpn-conf-proba-threshold

Minimal OVPN_CONF_LEVEL value considered positive [unsigned]
Default: 50

--ovpn-conf-threshold

Threshold for OVPN_CONF_LEVEL Detector [unsigned]
Default: 5

```
--wg-conf-proba-threshold
  Minimal WG_CONF_LEVEL value considered positive [unsigned]
  Default: 50
--wg-conf-threshold
  Threshold for WG_CONF_LEVEL Detector [unsigned]
  Default: 5
--ssa-conf-proba-threshold
  Minimal SSA_CONF_LEVEL value considered positive [unsigned]
  Default: 50
--ssa-conf-threshold
  Threshold for SSA_CONF_LEVEL Detector [unsigned]
  Default: 5
--tor-threshold
  Threshold for Tor Detector [unsigned]
  Default: 5
--ip-ranges-file
  Path to observed IP ranges file [string]
  Default: /opt/tunder/ipRanges.txt
--blocklist-file
  Path to blocklist file [string]
  Default: /opt/tunder/blocklist.txt
--blocklist-tick-interval
  Interval in seconds, in which blocklist file is checked
  for changes [unsigned]
  Default: 30
--blocklist-threshold
  Threshold for Blocklist Detector [unsigned]
  Default: 5
```

Args must be always passed separated by space:

OK: --tick-interval 10

FAIL: --tick-interval10

C.4 Malware Detector

MalDer - v1.2.1

Help

=====

Malware Detector (MalDer) is a C++ implementation of BOTA.

-h

Display help section [-]

Default: false

--bytes-threshold

Threshold for bytes anomaly detector [unsigned]

Default: 1000000

--ip-store-size

IP Store Size [unsigned]

Default: 1000000

--packets-threshold

Threshold for packets anomaly detector [unsigned]

Default: 10000

--ports-threshold

Threshold for port anomaly detector [unsigned]

Default: 1000

--ip-threshold

Threshold for IP anomaly detector [unsigned]

Default: 100

--ip-ranges-file

File with observed IP ranges [string]

Default: /opt/malder/ipRanges.txt

--ml-bridge-file

File with ML bridge [string]

Default: /opt/malder/bridge.py

--ml-model-file

File with ML model in pickle format [string]

Default: /opt/malder/ab.pickle

--cnc-buffer-size

Buffer size for CNC detector [unsigned]

Default: 10000

Args must be always passed separated by space:

OK: --tick-interval 10

FAIL: --tick-interval10

Contents of the Attached Archive

```
|_ docs/.....Doxygen documentation
|_ rpms/.....RPM packages
|_ sources/ ..... folder with source codes
|   |_ libwif/.....sources of Weak Indication Framework
|   |_ cryptominingDetector/.....sources of Cryptomining detector
|   |_ torDetector/ .....sources of Tor detector
|   |_ tunnelDetector/ ..... sources of Tunnel detector
|   |_ malwareDetector/.....sources of Malware detector
|_ tests/.....scripts and data used for testing
|   |_ cryptominingDetector/ ..... cryptomining specific
|   |_ torDetector/.....Tor specific
|   |_ tunnelDetector/.....tunnel specific
|   |_ malwareDetector/.....malware specific
|_ text/.....thesis-related sources
|   |_ overleafProject.zip.....project exported from Overleaf.com
|   |_ thesis.pdf.....this thesis in PDF format
```