FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Implementation of DNS over HTTPS Detector
Student:	Ondřej Hrdlička
Supervisor:	Ing. Karel Hynek, Ph.D.
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

Study computer network monitoring approaches. Get acquainted with DNS over HTTPS protocol and its detection possibilities. Implement the detector proposed in [1] as an NEMEA system [2] module, that can be deployed to real network monitoring infrastructure. Test the implemented detector using real-world network data and evaluate it using standard metrics (detection accuracy and processing speed).

[1] K. Jerabek, K. Hynek, O. Rysavy and I. Burgetova, "DNS Over HTTPS Detection Using Standard Flow Telemetry," in IEEE Access, vol. 11, pp. 50000-50012, 2023, doi: 10.1109/ACCESS.2023.3275744.
[2] T. Cejka, V. Bartos, M. Svepes, Z. Rosa and H. Kubatova, "NEMEA: A framework for network traffic analysis," 2016 12th International Conference on Network and Service Management (CNSM), Montreal, QC, Canada, 2016, pp. 195-201, doi: 10.1109/CNSM.2016.7818417.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 26 January 2024 in Prague.

Bachelor's thesis

IMPLEMENTATION OF DNS OVER HTTPS DETECTOR

Ondřej Hrdlička

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

Contents

List of Figures

List of Tables

List of code listings

I would like to express my sincere gratitude to my supervisor, Ing. Karel Hynek, Ph.D., for his guidance, suggestions, and expert advice throughout the entire process. Additionally, I am thankful to Ing. Tomáš Čejka, Ph.D., for introducing me to Karel and assisting with theme selection. Lastly, I am also grateful to my friends and family for their understanding and support during my studies.

Declaration

In Prague on May 14, 2024

Abstract

This bachelor thesis describes the creation of a DNS over HTTPS (DoH) network detector integrated within the NEMEA system. Network monitoring tools face challenges in detecting DNS queries since malware creators utilize encrypted DNS protocols. This work focuses on the implementation of the detector which combines three different detection methods—IP-based blocklist, machine learning classification, and active verification. Evaluation on a real-world dataset demonstrates the effectiveness of the detector in identifying DoH traffic with high accuracy while using only standard flow telemetry. Implementing the proposed detector within the NEMEA framework offers a deployable solution for high-speed networks, which can help prevent numerous security threats.

Keywords DNS, DoH, NEMEA, network monitoring, flow-based monitoring, DoH detection, machine learning

Abstrakt

Tato bakalářská práce popisuje tvorbu DNS over HTTPS (DoH) detektoru, který je integrován do systému NEMEA. Nástroje pro monitorování sítě čelí různým výzvám při detekování DNS dotazů, protože tvůrci malwaru využívají šifrované DNS protokoly. Tato práce se zaměřuje na implementaci detektoru, který kombinuje tři různé metody detekce—IP blocklist, klasifikaci pomocí strojového učení a aktivní ověřování. Vyhodnocení detektoru na datové sadě z reálné sítě ukazuje vysokou přesnost detektoru v identifikaci DoH provozu, a to pouze s využitím standardní flow telemetrie. Implementace navrženého detektoru v rámci NEMEA frameworku nabízí nasaditelné řešení pro vysokorychlostní sítě, které může pomoci předcházet mnoha bezpečnostním hrozbám.

Klíčová slova DNS, DoH, NEMEA, monitorivání sítě, monitorivání síťových toků, detekce DoH, strojové učení

Abbrevations

ALPN	Application Layer Protocol Negotiation
C2	Command and Control
DDoS	Distributed Denial of Service
DNS	Domain Name System
DoH	DNS over HTTPS
Gbps	Gigabits per second
HTTPS	Hyper Text Transfer Protocol Secure
IDS	Intrusion Detection System
IE	Information Element
IP	Internet Protocol
IPFIX	Internet Protocol Flow Information Export
ISP	Internet Service Provider
ML	Machine Learning
NEMEA	Network Measurements Analysis
PCAP	Packet Capture
SCTP	Stream Control Transport Protocol
SNI	Server Name Indication
SSH	Secure Shell
TLS	Transport Layer Security
TRAP	Traffic Analysis Platform
TTL	Time To Live
UDP	User Datagram Protocol
UniRec	Unified Record
VLAN	Virtual Local Area Network

Introduction

Domain Name System (DNS) is an essential part of today's internet, allowing the translation of human-readable domain names into numerical Internet Protocol (IP) addresses the computers can understand. This process enables users to access websites conveniently, send emails, and many other online activities. DNS protocol is one of the oldest network protocols, operating on a request-response scheme. For example, whenever a user wants to visit a website, he sends a message to a DNS server requesting to translate a certain domain name and gets an IP address as a response. When the protocol was designed, there were not many concerns about users' privacy or the potential misuse of the transmitted information.

Since the traditional DNS is unencrypted, meaning that anyone can see what is being sent, the transported information is susceptible to interception, analysis, and possible misuse by hackers or other intruders. Such vulnerabilities have raised concerns about the privacy and security of users on the internet. In response, the encrypted DNS protocol concepts were developed, and DNS over HTTPS (DoH) is by far the most common one. DoH secures the connection in a manner very similar to information protection by websites. It encapsulates the payload within a secure HTTPS connection, thereby enhancing security and privacy.

On the ability to inspect DNS packets rely several security systems, since its analysis can uncover or confirm numerous security threats, such as DNS data exfiltration or malware presence. Given that encrypted DNS protocols serve the same function as traditional DNS, the potential for misuse mirrors that of the original DNS protocol. The 2016 Cisco Annual Security Report [4] stated that 91.3% of malware families rely on DNS, primarily exploited for connecting to Command and Control (C2) infrastructure, data exfiltration, and traffic redirection.

The detection of encrypted DNS traffic is crucial for sustaining network security. By implementing the DoH detector proposed by Jeřábek et al. [16] and measuring its performance on a real-life dataset, we hope it will provide sustainable results. While other detectors and DoH detection methodologies already exist, they often lack the robustness and scalability required for real-world deployment. Integrating the proposed detector within the NEMEA framework was the motivation to overcome this issue and to provide a production-ready DoH detector, deployable on high-speed networks. According to Hynek [14], malware creators know the benefits of encryption and the problems associated with identifying encrypted DNS protocols and have started using these techniques in their creations.

Chapter 1
Theoretical Background

This chapter provides theoretical information about network monitoring possibilities and why it is an important aspect of sustaining the availability and security of any system or company. Network monitoring approaches, and specifically flow-based methods, are covered in Section 1.1. Following with Section 1.2, where Network Monitoring Analysis (NEMEA) system is introduced as a flow-based monitoring approach, which became highly recognized in recent years due to its flexibility and exceptional performance in high-speed networks. Finally, the DoH protocol is presented in Section 1.3 with all options and examples on how to transfer DNS data over HTTPS connection.

Networking is the backbone of modern information technologies, enabling devices and applications to exchange data between each other across the internet. To ensure efficient and secure communication between devices, data is usually sent via some network protocol—a set of rules prescribing how data is structured, transmitted, and received across network interfaces. Different protocols serve different functions, typically visualized through the OSI model. The model has seven different layers, ranging from the Physical Layer, responsible for the transmission of bits over hardware, to the advanced application-specific communication in the Application Layer. For instance, the Internet Protocol (IP) is responsible for routing data packets to their destination, thus making the network-to-network communication possible, or Transmission Control Protocol (TCP), which guarantees reliable transportation of packets.

Understanding and effectively managing network protocols is crucial to maintaining network security and availability. Effective monitoring helps network administrators and operators to ensure network reliability and to discover issues as they emerge. It involves observing and analyzing network performance to address any deviations from normal behavior, which could indicate issues like traffic congestion, poor network components, or security breaches. Intrusion Detection Systems (IDS) play a crucial role in this aspect since they are deployed within the network to detect unusual activities or known threats using some predefined rules or based on the observed events in network traffic. By identifying threats early, IDS enables quick response, therefore mitigating the risks and minimizing potential damage. But as cyber threats evolve, so does the need for more advanced detection techniques.

1.1 Network Monitoring

Network monitoring is a process of overseeing network traffic to ensure and maintain its availability and reliability, and to detect and address issues as soon as possible. The approaches of network monitoring can be generally divided into two categories [13]: active and passive. Active approaches insert new traffic into the network to conduct various types of measurements or to monitor the path that traffic takes to reach its destination. They can be implemented by tools such as Ping [1] or Traceroute [2]. Passive approaches, on the other hand, examine the existing network traffic generated by users, usually with the intention of detecting some anomalies based on the observed events.

One of the passive approaches is packet capture. This provides a really deep insight into the traffic, as entire packets are intercepted and investigated, but it requires a lot of resources to store and analyze high packet volumes. Another passive monitoring approach is flow export. This method is much more flexible to use in high-speed networks, as packets are accumulated into flows, holding mostly statistical characteristics of particular connection, and the flows are then exported for storage and analysis. A flow can be defined as a set of IP packets sharing some common properties primarily in packet header fields, such as source and destination IP addresses and port numbers [13].

The concept of flow export traces back to 1991, evolving significantly over the decades. Initially introduced by the Internet Accounting Working Group of the Internet Engineering Task Force (IETF), the idea faced early obstacles due to a lack of vendor interest and prevailing beliefs against internet traffic monitoring. However, by 1995, renewed interest led to the development of the Realtime Traffic Flow Measurement (RTFM) Traffic Measurement System, which offered a more flexible approach to traffic flow measurement [1]. Parallel to these efforts, Cisco developed NetFlow, a technology originating from flow-based switching, which became widely adopted for its ability to provide detailed traffic flow information. This led to the creation of NetFlow v5 [5] and, subsequently, the more advanced NetFlow v9 [8], offering enhanced capabilities such as support for IPv6 and Virtual Local Area Networks (VLANs). Recognizing the need for a standardized flow export protocol, the IETF chartered the IP Flow Information Export (IPFIX) Working Group in 2004, which built upon NetFlow v9 to develop IPFIX [6] protocol, which introduced some new features and became the internet standard by 2013 [13].

1.1.1 IP Flow

There are multiple definitions of IP flow in the internet community, but according to the one described by the IPFIX Working Group in [7], [21]: *"A Flow is defined as a set of IP packets passing an Observation Point in the network during a certain time interval. All packets belonging to a particular Flow have a set of common properties"*. An Observation Point is a place somewhere in the network from where IP packets can be observed and captured, e.g. interfaces of a router or shared medium like Ethernet-based LAN [7].

To determine whether a packet belongs to a certain flow, it has to satisfy all predefined properties of the flow. In the specification of IPFIX, these properties are called the flow key. They are for instance: source and destination IP addresses, source and destination port numbers, and used transport protocol. There are also other fields in the flow record, usually statistical characteristics used for analytical purposes. Table 1.1 shows standard flow record features in most available flow export devices and protocols [16].

1.1.2 Flow-Based Monitoring

Flow-based monitoring has become increasingly popular over the past years. Many network IDS rely on packet inspection, mentioned as another passive monitoring approach, however, the disadvantage of packet inspection is that it is too difficult, or even impossible, to perform at high-speed lines reaching hundred Gigabits per second (Gbps). Because of this, the flow-based approach has become more suitable for numerous security systems. In this alternative approach, only the IP flow records are investigated, thus consuming significantly fewer resources.

[1]https://linux.die.net/man/8/ping
[2]https://linux.die.net/man/8/traceroute

Flow Record
Source IP Address
Destination IP Address
Source Port
Destination Port
Transport Layer Protocol
Number of Packets
Number of Bytes
Time Start
Time End

Table 1.1 Standard Flow Record Features [16]

The typical flow monitoring architecture consists of multiple stages. According to Hofstede et al. [13], the first stage is Packet Observation, where packets are intercepted when passing an Observation Point. Initially, packets are read from the network line, usually by Network Card Interface [13], where they undergo several tests, like header checksums. The next step is to timestamp the packets. This is important because exact packet time is essential for many analytical purposes. Both of these steps are performed on every packet that passes the Observation Point. All following stages shown in Figure 1.1 are optional. Packet truncation involves selecting only those packet bytes that are necessary for subsequent stages. A great example of this is that flow exporters typically rely solely on packet header fields [13].

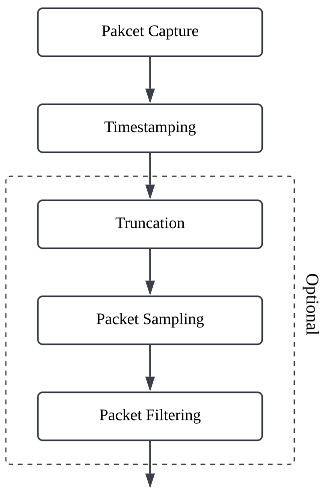


Figure 1.1 Packet Observation Architecture [13].

The purpose of packet sampling is to reduce the load on the following stages of the metering process. It means selecting only part of the observed packets to be processed and sent to the next phase. They are in general two types of sampling strategies [13]:

- *Systematic sampling*: Deterministic approach that chooses packets based on predefined rules. For example, select a packet every t seconds or a packet every n packets.
- *Random sampling*: This approach selects packets at random, which is generally more preferred, as it avoids problems the systematic sampling may introduce, especially in the case of periodic traffic.

Packet filtering is a deterministic process to separate all packets having a certain property from those not having it. There are again two main strategies [13]:

- *Property Filtering*: The packet is selected based on whether a certain field within the packet is equal to a predefined value or falls into a specified value range, e.g., IP addresses or number of transmitted packets.
- *Hash-Based Filtering*: This method involves applying a hash function to either the entire packet content or just some part of it. The output of the hash function is then compared to predefined values, efficiently selecting packets with common properties.

Packet sampling and filtering come into play when dealing with limited resources or on high-speed network lines, where capturing every packet would be impossible to perform. Both provide a trade-off between efficiency and the completeness of data. The final flow records might not include all packets but are designed to provide sufficiently accurate results.

Typically, there are more than one Observation Point within a network [7], such as multiple router ports. These points form an entity known as the Observation Domain, which serves as a scope where flow data is collected. Each Observation Domain must have a unique identifier, and this concept helps to organize, manage, and interpret flow data more efficiently, especially in complex network environments. Figure 1.2 shows an example of an architecture where three Observation Points contribute to a single Observation Domain.

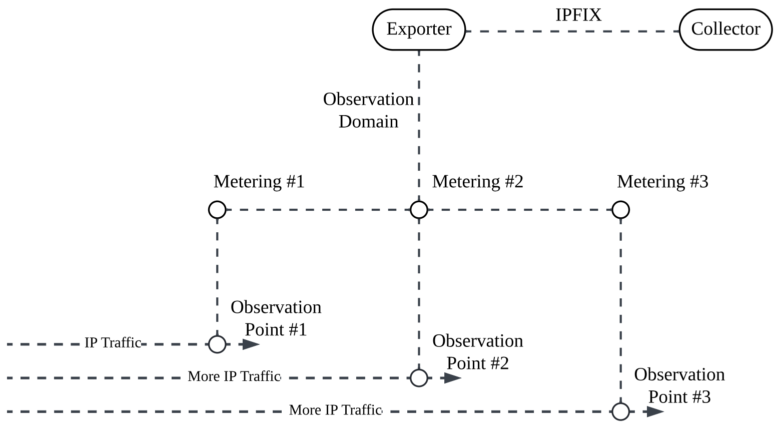


Figure 1.2 Flow-Based Monitoring Architecture Example

1.1.3 Flow Exporter

Flow exporter is a fundamental component of any flow monitoring system. It is where the packets are aggregated into flows and then exported for analysis. As described by Hofstede et al. [13], flow export contains two vital processes—the Metering process and the Exporting process—illustrated in Figure 1.3. The metering process is responsible for the aggregation of packets based on the packet elements defining the flow. The flows are stored in a flow cache, where they remain until the flow is considered terminated. There are several options for flow to expire provided later in this section.

The data that can be exported with flow records are called Information Elements (IE). These elements cover various network layers, enabling flow description from the Link Layer (L2) to the Application Layer (L7). However, common Information Elements are usually from the Network Layer (L3) or Transport Layer (L4) [13].

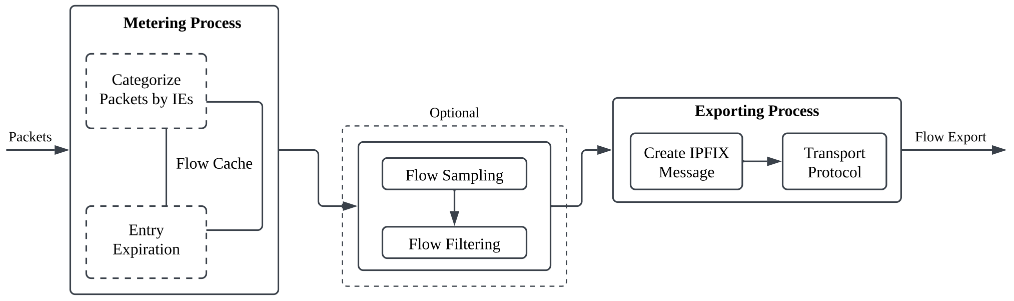


Figure 1.3 Metering and Flow Export Architecture [13]

Flow Caches

Flow caches act as temporary storage within a Metering Process for all active flows. The entries in the flow cache consist of Information Elements, which can be categorized as key or non-key fields. The set of key fields is used to decide whether a packet belongs to an already existing record in the flow cache or if there is a need to create a new one. According to Hofstede et al. [13], when a packet is captured, its flow key is hashed and compared with the existing flow entries. If a match is found, the specific flow entry is updated (e.g., byte and packet counters); otherwise, a new entry is created. As mentioned in Section 1.1.1, standard fields used for a flow key are usually source and destination IP addresses and port numbers. Non-key fields are typically used for collecting statistical characteristics, such as byte and packet counters. Since source and destination IP addresses are commonly part of the flow key, flows are unidirectional. However, in some situations, both directions are important, which leads us to bidirectional flow records, where special cache support is necessary for matching forward and reverse flows and for maintaining bidirectional records [13].

The capacity of the flow cache heavily depends on the memory resources available in the flow exporter, and it should be chosen considering the expected flow volume, the selected key and non-key fields, and expiration policies [13].

Flow Cache Entry Expiration

Flow entries are kept in the cache until they are identified as finished, after which they expire. The expiration is usually managed by the Metering Process, which is responsible for checking expiration rules violations. IPFIX suggests several reasons for considering a flow has expired [22]:

- A flow is considered as expired when it has been active for a specified period. This ensures that even long-running flows are reported periodically. In such cases, the flow entry is not removed from the cache, allowing the Metering Process to avoid creating a new record; instead, it just resets the counters and timestamps.
- A flow is considered as expired when no packets associated with it have been detected for a specified time period.
- A flow is prematurely expired if the device experiences resource constraints.

Other reasons for expiring cache entries can also be implemented, such as the observation of a TCP packet with a FIN or RST flag [13].

Exporting Process

When a flow expires, it is ready to be exported for further analysis. When using IPFIX as an export protocol, the data are sent within an IPFIX message, which begins with a 16-byte header including version number, message length, export time, and an observation domain ID, followed by one or more sets. These sets can be Template Sets, Data Sets, or Options Template Sets. Template sets outline the structure of the Data Records, Data Sets carry the exported Data Records (i.e., flow records), and Options Template Sets are used to send metadata to flow collectors—the flow key used by Metering Process [6].

After an IPFIX message is constructed, it is transmitted to a flow collector. IPFIX supports multiple transport protocols [6]—Stream Control Transport Protocol (SCTP), TCP, or User Datagram Protocol (UDP). SCTP is not widely used due to implementation challenges and limited support outside of Unix-like systems. TCP offers reliable transport across most platforms, making it the preferred option for exporting flows over the internet. Despite its lack of any congestion awareness, UDP is the most widely deployed protocol for flow export, as it creates almost no overhead to the system [13].

1.1.4 Flow Collector

Flow collectors play another crucial role in flow monitoring setups, as they are responsible for receiving, storing, and preprocessing flow data sent from flow exporters. This preprocessing may include the compression of flow data to reduce the storage requirements, aggregation to unify similar data, anonymization to protect user privacy, and filtering to exclude irrelevant data [13].

It is essential that the flow collector is compatible with the export protocol features used by the flow exporter, such as data encodings, transport protocols, and IEs. If the flow collector fails to support every exported element in the received data stream, there is a risk of data loss. Extra caution is necessary when using non-standard IEs, like application-specific or enterprise-specific IEs [13].

Data Storage Formats

The efficiency of flow collectors critically depends on the used data storage format, which directly impacts the speed and manner in which data is accessed and stored. There are two types of storage formats:

- *Volatile*: Volatile storage operates in-memory, offering very quick data access, and is typically used for immediate data processing or caching before it is written to persistent storage. Volatile storage is useful for on-the-fly analysis of flow data when only the results need to be archived.
- *Persistent*: Persistent storage, on the other hand, is designed for long-term data storage, providing commonly greater capacity at the cost of slower data access.

Persistent storage becomes necessary when the data has to be stored beyond the time needed for processing. Hofstede et al. [13] compared these three storage types:

- *Flat files*: Flat files, such as binary or text files, offer high-speed data access; however, they often come with limited querying capabilities. They provide better portability since many systems can read text-based flat files, but the lack of more sophisticated querying features can be a limitation for some use cases.
- *Row-oriented databases*: Row-oriented databases store data as rows within tables and serve as a common choice for Database Management Systems, like MySQL[3], or PostgreSQL[4]. They allow more complex queries but may require reading entire rows, even when only a subset of the data is needed.

[3]https://www.mysql.com/
[4]https://www.postgresql.org/

- *Column-oriented databases*: Column-oriented databases, such as FastBit[5], store data by columns, enabling the system to access only the necessary data fields, which can enhance the query performance.

The choice of storage format has a direct effect on disk space, insertion performance, portability across systems, and query efficiency. Flat files have a clear advantage in disk space and insertion speed but are less flexible in query operations. In contrast, row-oriented databases may consume more disk space due to indexing but offer more query flexibility. Columns-oriented databases provide a balance, offering better query efficiency and average disk space consumption. The selection between these storage types depends on the requirements of the flow monitoring setup [13].

1.2 NEMEA System

Network Measurements Analysis (NEMEA) is a flow-based, modular system for network traffic monitoring designed to analyze flow data continuously with minimal storage requirements. Monitoring probes (flow exporters) are responsible for observing the network and sending flow data to a central collector utilizing the IPFIX protocol. The collector stores the received flow records and sends them for analysis to the NEMEA system. The NEMEA system is comprised of independent, interconnected modules and is highly flexible and extensible, allowing users to implement and connect new modules easily. Each NEMEA module is an executable file that performs specific tasks, such as data preprocessing, data filtering, detection, or reporting [3].

Despite the distinct roles of each module, they all share the same NEMEA framework, which includes libraries implementing inter-module communication, data format, and common data structures and functions. Modules communicate with each other via TRAP (Traffic Analysis Platform) interfaces, an essential part of the framework, supporting unidirectional data streams. These data streams may include messages with flow data, computed analytical statistics, alerts of a detection module, or anything else [3].

Data can be exchanged over the TRAP interfaces in one of three supported formats—(i) JSON, (ii) unstructured data, and, most commonly, (iii) NEMEA's Unified Record (UniRec) binary format, which allows efficient and quick access to transferred data. The setup of interfaces of modules is done during startup, allowing users to select and connect modules as they desire. A NEMEA module does not require any specific knowledge about other modules but usually expects some predefined fields in the messages (e.g., IP addresses or packet counts) to operate properly. This means that a module can be connected to any other module but will work only in the presence of all essential data fields [3]. The figure shows the network monitoring setup with NEMEA, and important parts of the NEMEA system are captured in Figure 1.5.

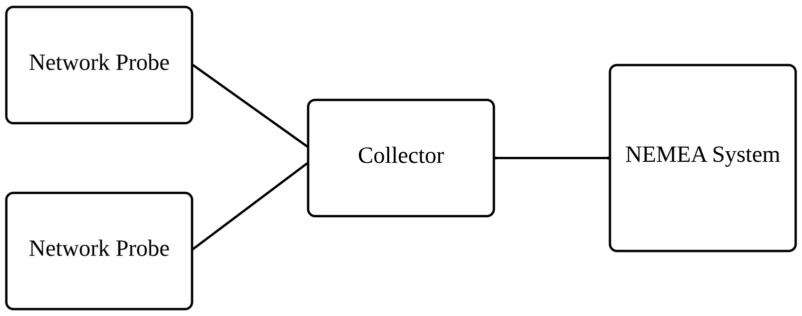


Figure 1.4 Network Monitoring with NEMEA [3]

[5]https://sdm.lbl.gov/fastbit/

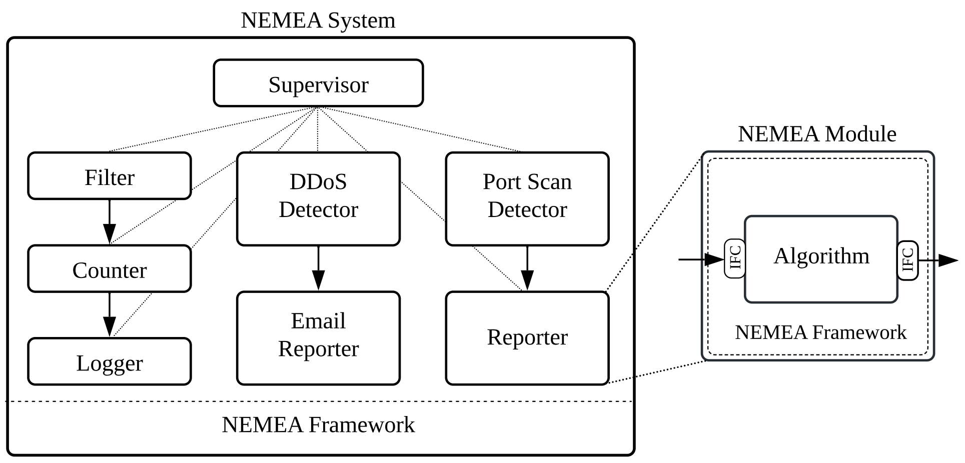


Figure 1.5 Parts of NEMEA System [3]

The core of the NEMEA system is typically some detector, responsible for identifying suspicious network traffic. The other modules include tasks like the exportation and storage of flow data, as well as pre-processing and post-processing of this data (filtering, aggregation, timestamping,. . .). The communication between these modules is done via TRAP interfaces, which implement functions for sending and receiving messages across the modules. These messages, typically UniRec messages, ensure an efficient and unified data format for the transmission. Finally, coordinating the entire NEMEA system is the Supervisor, a centralized management and monitoring tool that takes care of all running modules [3].

1.2.1 NEMEA System Deployment and Capabilities

NEMEA was first introduced in 2013, and initially developed for CESNET—the Czech National Research and Education Network operator, providing advanced network solutions and services to research and educational institutions in the Czech Republic. It was created to overcome the lack of existing tools for processing and analysis of L7 information since traditional flow exports provided visibility up to L4 only. This information can be necessary to detect some kinds of malicious traffic. NEMEA can handle high-speed network traffic links working at 100 Gbps, and has been deployed at CESNET since 2014. The system is capable of detecting various types of malicious traffic, including Distributed Denial of Service (DDoS) attacks, network scanning, or dictionary attacks on Secure Shell (SSH). The high performance of NEMEA is highlighted by its deployment in the CESNET2 network, demonstrating its ability to process large volumes of network data and detect numerous network incidents daily without significant resource consumption [3].

The NEMEA system deployed in the CESNET2 network can detect, on average:

- 110,000 horizontal port scans (76 every minute)
- 12,000 brute-force or dictionary attacks to log in to SSH (8.3 per minute)
- 2,400 DDoS attacks (mostly DNS and NTP (Network Time Protocol) amplification)

The numbers may vary at this time, but they demonstrate the capabilities of the NEMEA system when deployed on an intensively used network. It is rather the number of alerts than the number of actual attacks, but even then, the number of real attacks is still in the order of hundreds.

1.3 DoH Protocol

DoH is a relatively new protocol defined in RFC 8484 [12] in 2018 and leverages HTTPS to securely transmit DNS messages, thereby protecting domain name resolutions from potential eavesdropping. It is designed to operate in conjunction with HTTP/2 to maintain efficiency since it allows simultaneous transmission of multiple requests. The protocol employs features like multiple streams within a single TCP connection and frames multiplexing. Despite the RFC recommendations for HTTP/2 to enhance the protocol's performance, it is noted that some DoH resolvers only support HTTP/1 [16].

DoH client can encode a DNS query into an HTTP request using either the HTTP GET or POST method. The queries are then sent to a resolver in the body of the HTTP POST method encoded using DNS wire format, defined in RFC 1035 [19], or within the query string of GET method, encoded with base64url [12].

The flexibility and ambiguity of DoH specifications open the door for various server behaviors. According to [15], the shape of DoH traffic can significantly vary depending on the client-server configuration and implementation. Presently, DoH is enabled by default in all major internet browsers, including Chromium-based [6] or Mozilla Firefox [7] [16].

1.3.1 DoH Request

When the POST method is selected, the DNS query is embedded within a message body of the HTTP request, with the `content-type` request header field specifying the message media type. In general, POST requests are more compact than their GET equivalents. Figure 1.6 shows the POST DoH example query for `www.example.com`. The 33 bytes are the DNS message in DNS wire format [19].

```
:method = POST
:scheme = https
:authority = dnsserver.example.net
:path = /dns-query
accept = application/dns-message
content-type = application/dns-message
content-length = 33

<33 bytes represented by the following hex encoding>
00 00 01 00 00 01 00 00  00 00 00 00 03 77 77 77
07 65 78 61 6d 70 6c 65  03 63 6f 6d 00 00 01 00
01
```

Figure 1.6 DoH POST query example [12]

The GET request for the same URL can be seen in Figure 1.7. The body of the message is empty and the queried domain is sent as a value of `dns` variable in the URL, encoded with base64url format.

1.3.2 DoH Response

DoH responses should only be the type of `application/dns-message` as stated by Hoffman et al. in RFC 8484 [12], but it is not impossible that other formats will be presented. The payload

[6]https://www.chromium.org
[7]https://www.mozilla.org

```
:method = GET
:scheme = https
:authority = dnsserver.example.net
:path = /dns-query?dns=
   AAABAAABAAAAAAAAA3d3dwdleGFtcGxlA2NvbQAAAQAB
accept = application/dns-message
```

Figure 1.7 DoH GET Query Example [12]

for this media type is a message in the DNS wire format. In Figure 1.8, you can see a response example for requesting `www.example.com` domain. The response contains an answer with the IP address of `2001:db8:abcd:12:1:2:3:4` and Time To Live (TTL) of 3709 seconds.

```
:status = 200
content-type = application/dns-message
content-length = 61
cache-control = max-age=3709

<61 bytes represented by the following hex encoding>
00 00 81 80 00 01 00 01  00 00 00 00 03 77 77 77
07 65 78 61 6d 70 6c 65  03 63 6f 6d 00 00 1c 00
01 c0 0c 00 1c 00 01 00  00 0e 7d 00 10 20 01 0d
b8 ab cd 00 12 00 01 00  02 00 03 00 04
```

Figure 1.8 DoH Response Example [12]

1.3.3 DoH using JSON format

Additionally, some DoH resolver providers, such as Cloudflare or Google, also support the option to transfer DNS data using JSON format. JSON formatted queries are sent as an HTTP GET request, with the DNS query encoded in the URL. As said on Cloudflare's official website [23], the client should include an HTTP accept header with a type of `application/dns-json`, to indicate its ability to accept a JSON response. The example is taken from Cloudflare documentation [23] and the DoH request can look like the one on Figure 1.9.

```
GET /dns-query?name=example.com&type=AAAA HTTP/2
Host: cloudflare-dns.com
accept: application/dns-json
```

Figure 1.9 DoH JSON Request Example [23]

Figure 1.10 shows an example of a DoH response for the previous request in JSON format.

1.3.4 Detection of DoH

The detection of DoH is challenging, as it utilizes HTTPS protocol to encapsulate and transfer DNS data, thus blending in with regular HTTPS traffic. The necessity of the protocol's detection was first recognized in 2020 by Bumanglag and Kettani [2], which surveyed the influence of mass

```
HTTP/2 200
server: cloudflare
content-type: application/dns-json
content-length: 210

{
  "Status": 0,  // NOERROR - Standard DNS response code
  "TC": false,  // Whether the response is truncated
  "RD": true,   // Whether Recursive Desired bit was set
  "RA": true,   // Whether Recursion Available bit was set
  "AD": true,   // Whether response data was validated with
     DNSSEC
  "CD": false,  // Whether the client asked to disable DNSSEC
  "Question": [
    {
      "name": "example.com.",
      "type": 28        // record type AAAA - IPv6 address
    }
  ],
  "Answer": [
    {
      "name": "example.com.",
      "type": 28,
      "TTL": 1726,
      "data": "2606:2800:220:1:248:1893:25c8:1946"
    }
  ]
}
```

Figure 1.10 DoH JSON Response Example [23]

DoH adoption. In late 2022, García et al. [11] examined the efficiency of using a blocklist for DoH detection, with the conclusion that due to the presence of numerous small and private DoH resolvers, IP-based blocklisting is not sufficient for reliable detection. They also stated that 94% of the public DoH resolvers are unknown to the internet community, and the number of resolvers has increased almost 5 times during the 2021–2022 period.

Related Work

Several studies employed machine learning (ML) techniques to learn the shape of DoH traffic by analyzing extended flow characteristics. Vekshin et al. [24], for instance, extracted 18 features from extended flow records and achieved a high accuracy of 99.6%. Similarly, MontazeriShatoori et al. [20] and Bandaki [18] have also employed machine learning with extended flow records and time-related features to distinguish DoH traffic from regular HTTPS, with even higher accuracy results.

Even though all mentioned techniques achieved pretty good results, they are limited in terms of deployment since they are either packet-based or require additional data sources—individual packet information or time-related features. These approaches use features like the median and variance of packet sizes, which require more computational and memory complexity. These disadvantages prevent them from deployment on high-speed networks or monitoring infrastructure with limited hardware resources.

Fest et al. [10] proposed a DoH detector using a neural network that relied on the standard flow telemetry. Nevertheless, it had a lower accuracy (94.4%) compared to other proposals. Therefore, Jeřábek et al. created a novel detector [15]. It uses standard flow telemetry, supported by almost all flow monitoring devices while attaining similar or even better results. Furthermore, the method relies only on features that can be computed on running sequences, enabling its deployment across any flow monitoring infrastructure, including high-speed Internet Service Provider (ISP) networks.

Chapter 2

Dataset and Analysis

This chapter introduces the dataset created by Jeřábek et al. [17]. It was used for the ML classifier's training phase and also to visualize and examine the characteristics of DoH in contrast to other HTTPS traffic. This should help us understand future classifier's predictions and also provide deeper insight into DoH protocol.
The dataset was created in late 2021 and contains comprehensive data samples of DoH and regular HTTPS traffic. Section 2.1 briefly outlines the methods used when creating the dataset and the types of data it includes. This is followed by Section 2.2, where, through a series of graphs, we demonstrate some of the key differences of DoH.

2.1 Data Collection

The dataset contains labeled HTTPS traffic, either DoH or non-DoH, focusing on providing a wide array of DoH communication traffic from various DoH services to represent the diversity of real-world DoH implementations. The dataset is a collection of smaller datasets aggregating to around 430 GB of raw binary data, derived from two environments—(i) real-world DoH and HTTPS traffic from a large ISP backbone network with about half-million users and (ii) generated DoH and HTTPS traffic from simulated and controlled system. The goal was to provide datasets with heterogeneous DoH and non-DoH traffic [17].

2.1.1 Real-Word Data

The real-world datasets were captured on the CESNET2 network, a large-scale ISP infrastructure connecting several Czech educational and research institutions, hospitals, and even some government offices. Data was captured by utilizing six strategically located monitoring points across the CESNET2 perimeter in three locations—Prague, Brno, and Ostrava—each of them linked to one or more 100 Gbps network lines. The packet capturing was performed between June and October 2021, including different days and times. After the packets were captured, they were merged into Packet Capture (PCAP) files. However, these PCAP files could not be used directly but had to go through a comprehensive post-processing phase [17].

Deduplication of packets is one of the post-processing tasks. There is a chance that the packet is captured on multiple observation points since the same packet may traverse multiple paths before reaching its destination. Another critical step was data anonymization. This involved masking the IP and MAC addresses—each address (except the IP addresses of DoH resolvers) was replaced by part of its SHA256 hash with secret salt—and also substituting the packet payload with generic data. The process was performed automatically and was carefully designed to meet ethical standards while preserving the data's value for research purposes. The PCAP files were then used to create IP flows using the ipfixprobe[1], an open-source flow exporter. Flows were extended for Transport Layer Security (TLS) information obtained from TLS handshake [17].

More information about the capturing filter, packet deduplication, anonymization, and flow export methods used to create this dataset can be found in Jeřábek et al. [17].

2.1.2 Generated Data

The generated datasets were constructed within a controlled laboratory environment to produce a diversity of DoH and regular HTTPS traffic scenarios. Jeřábek et al. [17] had used two web browsers for this, Firefox and Chrome, primarily because of their advanced DoH implementations. The traffic generation involved browsing sessions where each browser visited thousands of websites, using 16 different DoH resolvers to cover a broad spectrum of DoH implementation behaviors. This approach ensured that the captured samples contained distinct traffic patterns since the shape of DoH communication can significantly differ because of the server and browser configuration [15].

Both browsers, Firefox and Chrome, fetched in total of 32,000 different websites. The traffic generation process consisted of opening the browser and fetching the website, timeout, waiting for the website to load, and then closing the web browser. The captured PCAP files were processed the same way as in the case of real data, using the ipfixprobe to create TLS extended flow records.

2.1.3 Final Dataset

The final dataset is merged from the real-world captured data and the generated data and contains nearly 15 million flow records. Each record includes detailed metadata like timestamps, source and destination IP addresses (anonymized), port numbers, and transport protocol identifiers and is also enriched with the TLS-specific attributes extracted during the connection establishment phase.

The comprehensive nature of the dataset enables the exploration of various aspects of DoH and HTTPS traffic, offering valuable insights into the dynamics of encrypted internet communications.

2.2 Data Analysis

This section explores the characteristics of DoH traffic compared to traditional HTTPS, using graphs to highlight some of the key features. These insights contribute to a better understanding of the final feature set for the ML classifier, Section 3.4. Jeřábek et al. found several observations about characteristics of DoH traffic, as highlighted in [15] and [16], and I wanted to demonstrate them on the dataset presented in the previous section 2.1. The characteristics can be summarized with the following observations:

Observation 2.1*:* DoH typically transmits less data in requests and responses than in regular HTTPS.

Observation 2.2*:* DoH connections are more symmetrical in the meaning of transferred packets and bytes in each direction.

Observation 2.3*:* The packet size variance of DoH is lower than in other HTTPS traffic.

Observation 2.4*:* DoH in the browser creates multiple streams over the same HTTP/2 connection and uses multiplexing for faster DNS resolution.

Observation 2.1 is captured in Figure 2.1, showing that the average DoH packet size is significantly smaller—almost three times less—compared to regular HTTPS traffic. The average

[1]https://github.com/CESNET/ipfixprobe

size of outbound DoH packets aligns well with the information from [9], that DNS messages are typically around a hundred bytes.

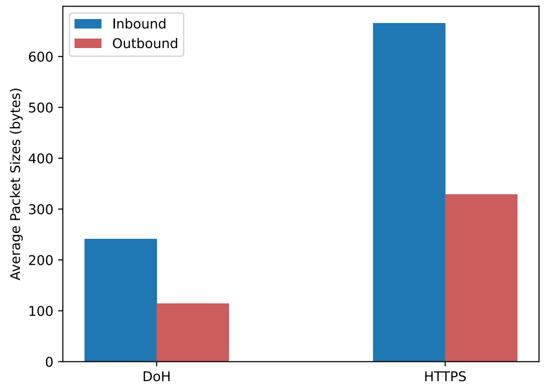


Figure 2.1 Average Packet Sizes by Protocol and Direction

The symmetry of DoH traffic in terms of transmitted packets is depicted in Figure 2.2. It can be clearly seen that DoH communication is almost equally balanced in the manner of inbound and outbound packets. With HTTPS, it is normal that there is significantly more inbound traffic. For instance, a web page request is usually small but receives a larger response, including data like images, scripts, and ads, which require more packets.

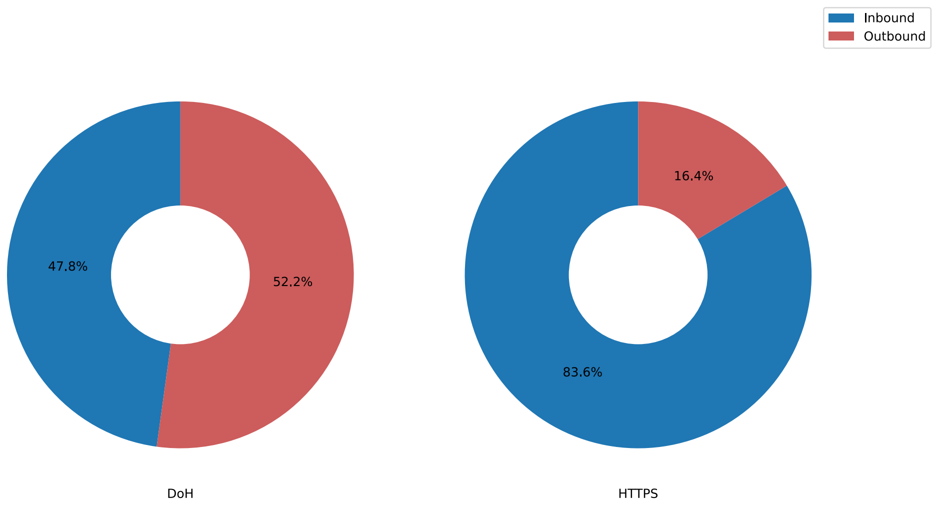


Figure 2.2 Ratios of Transmitted Packets by Protocol and Direction

In terms of transmitted bytes, the number of transmitted bytes in HTTPS was so much bigger than in the case of DoH that it had to be scaled (see the difference of the right and left y-axis in Figure 2.3, as the HTTPS axis is hundred times the DoH axis). Moreover, kilobytes were compared rather than bytes for better readability. Even though it is not as symmetric as in the term of packets, when compared with HTTPS, the difference is significant.

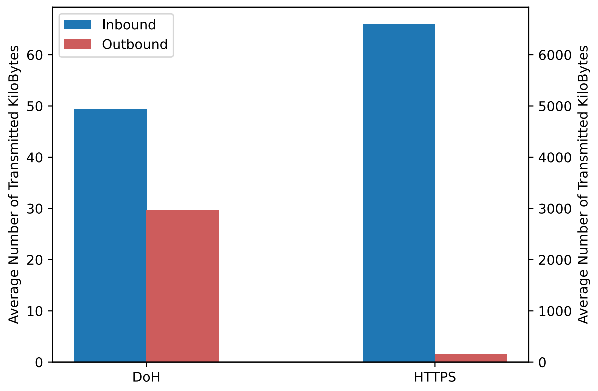


Figure 2.3 Average Number of Transmitted Kilobytes by Protocol and Direction

Figure 2.4 reflects the Observation 2.3, where you can see that the mean packet size variance in DoH communication is more than 2 times lower than in HTTPS. Variance is a measure of dispersion, meaning how far the values are spread out from their average value. In conclusion, packet sizes within the DoH protocol do not vary as much as in other HTTPS communication.

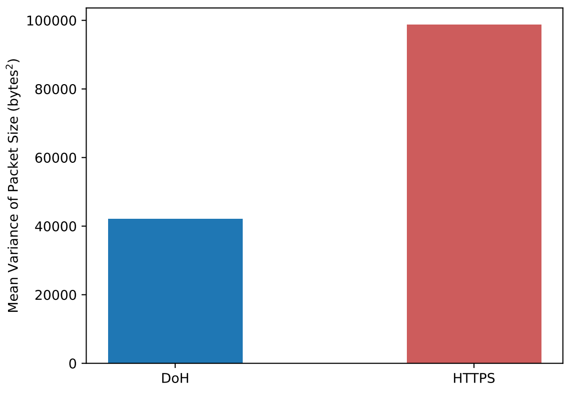


Figure 2.4 Mean Packet Size Variance by Protocol

The final graph illustrates the mean time between packets within a flow, Observation 2.4. Since DoH typically utilizes multiplexing of HTTP/2 to send multiple DNS queries in rapid succession without waiting for previous responses, it can lead to more frequent packet arrivals. This can result in faster overall resolution time as DNS requests can be transmitted at the same time, and responses can be received out of order. Figure 2.5 shows that the mean time between packets is lower in the DoH protocol for both directions.

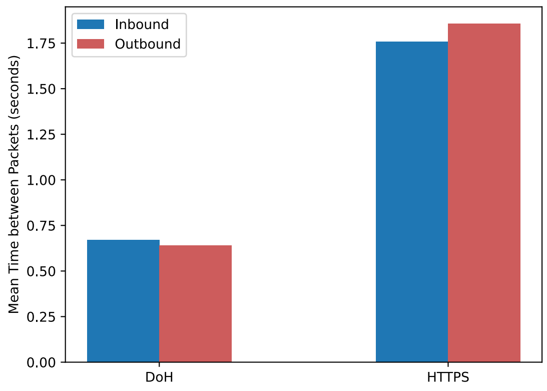


Figure 2.5 Mean Time between Packets by Protocol and Direction

Chapter 3

Design

This chapter outlines the pipeline of the DoH detector proposed by Jeřábek et al. [16]. It details the procedural architecture of the detector, describing how each component contributes to the overall detection mechanism and ensures the detector robustness. By providing insights into the methodology and strategic design of the detector, it lays out the groundwork for understanding the innovative approach to detecting DoH traffic. Individual detector parts are introduced through Sections 3.2–3.6.

3.1 Scheme

The detector comprises three different types of detection—IP-based, ML-based, and active probing. It is designed to work with traditional flow telemetry data with standard flow features but requires bidirectional flow records. If the used flow exporter does not support this feature, it can be achieved with flow stitching—a process that can construct from two unidirectional flow records a bidirectional one. An example of bidirectional flow can be seen in Table 3.1. The detector utilizes these three detection types since relying only on statistical machine learning methods would generate a high number of false positives [16].

Given that the active probing phase is a resource-consuming task, the detection system employs an ML classifier to select only suspicious flows worth verification. Additionally, the detection pipeline includes a feed-forward loop, where the verification step generates a blocklist/allowlist, which is then used by the IP-based module. By employing these three distinct classification approaches, the detector overcomes the limitations of each of them: the records in the IP list are time-sensitive and continuously updated through the verification step; the ML classifier miss rate is mitigated by active verification; and the resources needed by active verification are minimized by the IP list and ML module [16].

The detection pipeline is depicted in Figure 3.1. Further information about each step is provided in the following sections.

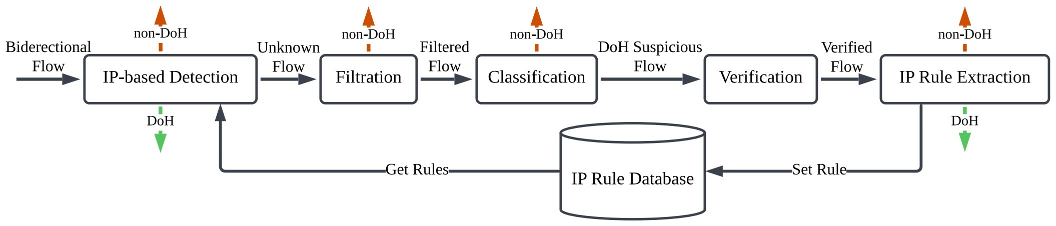


Figure 3.1 Detection Pipeline of the Proposed DoH Detector

Flow Record
Source IP address
Destination IP address
Source Port
Destination Port
Transport Layer protocol
Number of packets $S \rightarrow C$
Number of bytes $S \rightarrow C$
Number of packets $C \rightarrow S$
Number of bytes $C \rightarrow S$
Time start
Time end

Table 3.1 Bidirectional Flow Record Example. ($S \rightarrow C$ and $C \rightarrow S$ abbreviations stand for Server-to-Client and Client-to-Server directions)

3.2 IP-Based Detection

The IP-based detection utilizes previously gathered information about destination IP addresses, which is maintained by the IP Rule database and IP Rule Extraction module. By observing the destination IP address field, it can quickly detect DoH traffic. The flow is forwarded to the next phase when there is no prior knowledge about the destination IP address [16].

In order to improve the accuracy of detection results, we decided to add another parameter for decision-making, than observing just the IP address. The detection relies also on the hostname used when establishing the connection, part of a TLS SNI (Server Name Indication) extension. The SNI parameter is used when multiple hosts share one IP address. This can be done with virtualization and is a common thing these days. With this approach, we minimized the possibility of incorrect detection, and it should help to achieve better and more precise results when using a simple list-based detection.

3.3 Filtration

The filtration phase selects only those records with a predefined set of properties to reduce the number of flows for the ML classifier. Given that the DoH connection is very similar to any other HTTPS connection, it includes a TCP handshake, TLS handshake, HTTP/2 preface, application data transfer, and TCP termination, as shown in Figure 3.2. The presence of numerous handshakes and HTTP/2 preface packets, in contrast to the small amount of DoH data, significantly complicates stable detection. According to Hynek [14], short DoH connections, especially the single-query DoH, are very challenging to detect because they are very much alike to other short API calls. Therefore, it is essential to determine the minimal number of packets within a flow to distinguish DoH traffic from other HTTPS traffic reliably. The flow is instantly labeled as non-DoH when is too short for a reliable DoH detection. Other flows are forwarded to the next step [16].

Additionally, we also filter flows where the number of incoming or outgoing packets is equal to zero since these IEs are used in a division when computing features for the ML classifier.

3.3.1 Minimal Number of Packets Threshold

This section very briefly describes the process of determining the minimum number of packets needed to reliably distinguish DoH from non-DoH traffic performed by Jeřábek et al. [16]. Two

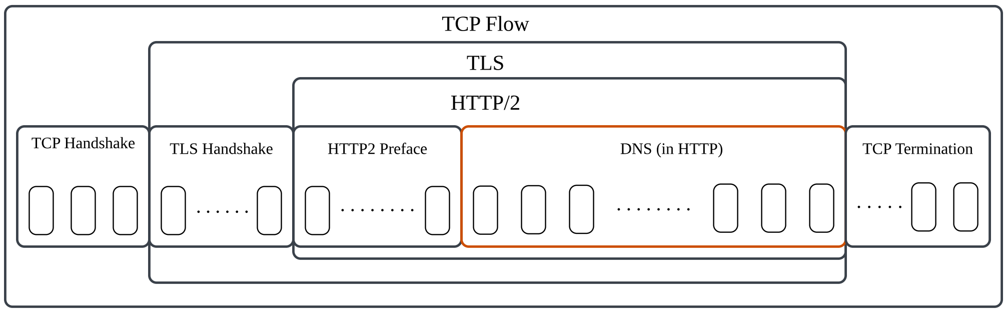


Figure 3.2 DoH Flow Example [16]

approaches were employed: finding the best threshold value during the hyperparameters tuning phase and unsupervised clustering with the K-Means algorithm. By incrementally increasing the packet threshold and retraining the XGBoost[1] classifier, the model's accuracy, as expected, was improved with a higher packet count, stabilizing around 116 packets. The K-Means clustering method (with $k = 2$; the number of classes) was applied, and a purity score was calculated for the formed clusters, resulting tin hat flow must contain at least 112 packets to achieve 9a 0% purity index of both clusters [16].

Jeřábek et al. [16] also examine DoH connections for the top 10,000 visited websites from the Majestic Million dataset. On average, these websites referenced another 20 unique domains to load all the website content (images, scripts, advertisements). When measured in packets, it gives roughly 120 packets per DoH flow for one website visit [16].

The experiments indicated a minimum of 112–116 packets is required for reliable DoH detection, although the threshold was set to ≥ 120 packets since this number corresponds to an average single website visit.

3.4 Classification

The classification phase utilizes a machine learning model to identify DoH-suspicious flows, which should then be actively verified. The ML classifier depends completely on the statistical characteristics of the flows. ML models make decisions based on a series of data known as features, which are measurable properties of individual instances.

The process begins with the preparation of the dataset. The dataset comprises many samples, each with a set of features and a corresponding label. The label represents the category or group that the model will learn to predict—DoH and non-DoH traffic in our case. The labels enable the algorithm to learn patterns and distinguish between the categories.

3.4.1 Used Features

The process of selecting features for the classifiers outlined here is derived from the methodology used by Jeřábek et al. [16], and is provided as a summary to contextualize the final feature set presented in Table 3.2.

The extraction of features from incoming flow data is one of the most important tasks when designing any ML model since it directly impacts the model's performance. Some of the shape characteristics of DoH traffic are demonstrated in Section 2.2.

Rather than using flow information elements directly, the approach involved generating all possible pairs through a division, resulting in a set of 21 features. Following this, a feature reduction process was applied by computing the pair-wise Pearson correlation coefficient. This

[1]https://xgboost.ai/

step removed features with a really high correlation, a correlation coefficient higher than 0.9, resulting in 4 final features [16].

ID	**Feature Name**
1	mean payload size $S \to C$
2	mean time between packets $S \to C$
3	mean payload size $C \to S$
4	num of $C \to S$ packets to packets ratio

Table 3.2 Final Feature Set for the ML Classifier

The final feature set reflects most of the traffic observations from Section 2.2. The most important feature #1, along with feature #3, captures DoH traffic behavior from observation 2.1. The feature #4 reflects the flow symmetry from observation 2.2, and observation 2.4 is captured by feature #2.

3.4.2 Machine Learning Classifier

Jeřábek et al. [16] applied numerous classification algorithms to the prepared dataset, such as Random Forest, K-Nearest Neighbors, Naive Bayes, and also popular boosting algorithms XGBoost and AdaBoosted Decision Trees. To determine the best performing one, standard metrics for unbalanced datasets were used, and XGBoost resulted as the best one.

During my work, I particularly focused on the XGBoost classifier not only because it was the best performing one in [16], but also for its:

- Efficiency: It is known for its execution speed and performance, which is a vital aspect in network monitoring when dealing with large volumes of data. It also utilizes a more regularized model formalization to control over-fitting, which makes the model more robust.
- Extensive Parameters Tuning: XGBoost provides a comprehensive range of tunable hyperparameters allowing precise optimization to suit specific datasets and detection tasks. This can significantly improve detection rates and reduce false positives.
- Performance & Popularity: XGBoost models have exceptional results in comparison to many other classification algorithms and have become a standard tool, especially in areas where the highest levels of accuracy are required.

XGBoost, or Extreme Gradient Boosting, is a sophisticated and powerful algorithm that builds a series of simple models (trees) to create a more complex and accurate model. It starts with a very simple tree, initially some basic conditions on the used features, and evaluates how well it did. In the next step, it pays attention to the instances that were predicted incorrectly. Each new tree is built to correct the mistakes of the previous one and iteratively enhances the model's accuracy. Once enough trees are made, XGBoost combines them all to create the final model and to make decisions. It also has built-in methods to prevent overfitting. Overfitting means that the trained machine becomes overconfident and performs really well on the training dataset, but poorly on new, unseen instances. There are numerous parameters, called hyperparameters, which help you adjust the learning process of the XGBoost algorithm, including the number of trees, their depth, and more.

Hyperparameters Tuning

Hyperparameters enable you to fine-tune the algorithm specifically to your dataset. To achieve the best results, you typically need to find and select the right hyperparameters, a process known as the hyperparameter tuning phase.

Jeřábek et al. [16] did not spespecifyich hyperparameters were used. In this work, Optuna[2], an open-source hyperparameter optimization framework, was employed to automate the hyperparameter search. The best combination was identified using a five-fold cross-validation procedure, selecting the hyperparameters with the highest average *F1 score*. The final set of tuned hyperparameters contains:

- `eta` (learning_rate): This scales the contribution of each tree to the whole model. When `eta` is close to 0, it reduces the model's learning rate and requires more boosting rounds, often leading to better generalization.
- `gamma` (minimum_split_loss): Specifies the minimum loss reduction required to make a further partition on a leaf node of the tree. The larger the `gamma` is, the more conservative the algorithm will be.
- `max_depth`: Defines the maximum depth of a tree. Higher values will make the model more complex and more likely to overfit, as higher depth will allow the model to learn relations very specific to a particular sample.
- `subsample`: This parameter controls the fraction of the training set that is randomly sampled and used to train individual trees.
- `colsample_bytree`: This sets the fraction of features used to train each tree.
- `lambda`: This is the L2 regularization term on weights. Higher values of `lambda` will make the model more conservative.
- `alpha`: This is the L1 regularization term on weights. Higher values of `alpha` will make the model more conservative.
- `n_estimators`: Represents the number of trees the model will consist of. More trees can give a more complex model, which can improve training accuracy, but there is a risk of overfitting.
- `tree_method`: The `tree_method` parameter specifies the algorithm used when constructing the trees. There are three methods available, but the `exact` method, which was selected, enumerates all possible splits on all features. It is the most exhaustive and accurate method, ensuring that the best-split point is found at each node. It can be very slow when building and training the model, especially with large datasets, but should provide the best results.

3.5 Verification

The next phase in the detection process involves active verification. At this stage, any record entering this module has been identified by the classification module as a potential DoH flow. Now it needs to be verified, whether the server, destination address, offers DoH resolution or not. The active probing module queries the server for the `example.com` domain using all possible DoH request methods, described in Section 1.3. In shared infrastructures, servers typically require a domain name to establish the connection to the correct endpoint.

Very similar to the IP-list detection, the active probing module was enriched to also use a domain name captured from a TLS handshake. This ensures more precise host verification and fits with the IP rules stored in the database. By storing the results in the IP Rule database, the module queries each target just once within a predefined time period, thereby minimizing the resource usage for both, the target and the module. Furthermore, whenever the DoH resolver rule (i.e., blocklist rule) is about to expire, the module regularly checks it to enhance the efficiency of the database.

[2]https://optuna.org/

3.6 IP Rule Extraction

The final step in the detection pipeline is IP Rule extraction, which handles the results from the active probing module. It stores both results, DoH verified and non-DoH verified, in the database, which is utilized by the IP-based detection step. This limits the number of flows advancing to the ML-based detection and active verification phase, thus enhancing the detector's throughput and efficiency—substantial aspects in any network monitoring system.

Chapter 4

Implementation

The detector is implemented as three inter-connected NEMEA modules—IP-based detection & filtration module, ML classifier module, and active verification & rule extraction module; all written in Python. They are built upon the pytrap[1] library, which is a wrapper for libtrap and UniRec libraries that allows you to develop NEMEA modules in Python. Libtrap defines the communication interfaces for message transfer between modules, and the UniRec library is the implementation of a flexible and efficient data format of flow records, both originally written in C.

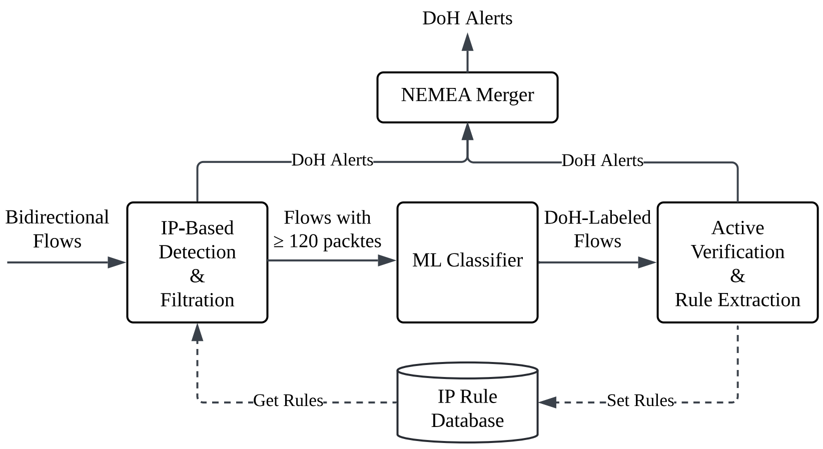


Figure 4.1 Implementation Scheme of NEMEA Modules

Figure 4.1 depicts the implementation scheme of the detector. The detector utilizes Redis[2] central database for storing IP rules, used for managing blocklist and allowlist in the IP-based detection module. NEMEA Merger[3] is used to merge UniRec messages containing DoH alerts from two input interfaces into one output interface.

4.1 IP-Rule Database

Redis was chosen for managing the block and allow rules utilized by the IP-based detection module. Redis is an in-memory database system designed for high performance and speed,

[1]https://github.com/CESNET/Nemea-Framework/tree/master/pytrap
[2]https://redis.io/
[3]https://github.com/CESNET/Nemea-Modules/tree/master/merger

exceptionally fast for read and write operations. Redis also supports setting expiration time on keys you insert, which is a great feature when managing rule lists, as you can conveniently set the rule expiration and let Redis take care of it. When properly configured, Redis does offer data persistence options, but it is primarily an in-memory database and was implemented as one. While it is not designed for complex queries, this was not a limiting factor since our read operations in a blocklist/allowlist scenario are simple. Other database systems, like MySQL[4] or SQLite[5], were also considered. While both have their strengths, especially in complex querying, the operational speed and efficiency of Redis outperform them and thus was selected as the best option for us.

Each module that needs to communicate with the database has a class called `RedisManager`. The class starts with connecting to the database in the constructor, as you can see on Listing 4.1. The parameters used in the `Redis` method are obtained from the central configuration file, discussed later in Section 4.5.

```
def __init__(self):
    try:
        self.db = redis.Redis(host=db_host, port=db_port, db=
            db_number, password=None)
        self.db.ping()
    except Exception as e:
        logger.error(f'Failed to connect to Redis: {e}')
        print("Failed to connect to Redis DB ")
        sys.exit(1)
```

Code listing 4.1 RedisManager Constructor

The methods the class has implemented differ based on the module's purpose, but in both cases, the class utilizes the redis-py[6] library, which is a Redis client for Python. `RedisManager` in the IP-based detection module has only one method to retrieve IP rules from the database. An example of an IP rule is illustrated in Listing 4.2 and has the form of "`{rule_type}:{IP},{SNI}`", where the `rule_type` can be either `block` or `allow`. The method `get_ip_rules` for obtaining the rules takes two arguments—`self`, which is a reference to the class that called the method; and `type`, which specifies the `rule_type`. The method is shown in Listing 4.3 and is pretty straightforward, as it iterates through the database and returns a set filled with rules of a certain `rule_type`.

```
1) "block:94.140.14.14,dns.adguard.com"
2) "block:8.8.4.4,dns.google.com"
3) "block:8.8.8.8,dns.google"
```

Code listing 4.2 IP Rule Examples

```
def get_ip_rules(self, type):
    pattern = f"{type}:*"
    rules = set()
    for key in self.db.scan_iter(pattern):
        rule = key.decode('utf-8').split(':')[1].split(',')
        rules.add((rule[0], rule[1]))

    return rules
```

Code listing 4.3 Method for Getting IP Rules

`RedisManager` in the active verification module has a method called `set_rule` for setting new rules. It takes four arguments—`self`, `ip`, `tls_sni`, and `rule_type`—all of them with the same

[4]https://www.mysql.com/
[5]https://www.sqlite.org/
[6]https://github.com/redis/redis-py

meaning as in the previous case. The method is presented in Listing 4.4. The `setex` method sets the key that expires in `rule_validity` seconds, which is also obtained from the central configuration file.

```
def set_rule(self, ip, tls_sni, rule_type):
    key = f"{rule_type}:{ip},{tls_sni}"
    self.db.setex(key, rule_validity, 'active')
```

Code listing 4.4 Method for Setting New IP Rule

The class also has two other methods for capturing and handling expired rules. Both methods are shown in Listing 4.5. The first method listens for any message and if it contains an expired rule, the second method is called. The rule is parsed and eventually inserted into the queue, where it waits until it is checked again.

```
def listen_for_expiration(self):
    pubsub = self.db.pubsub()
    pubsub.psubscribe('__keyevent@0__:expired')

    while True:
        message = pubsub.get_message()
        if message and message['type'] == 'pmessage':
            expired_key = message['data'].decode('utf-8')
            self.handle_expiration(expired_key)

def handle_expiration(self, key):
    list_type, ip_name = key.split(':')
    ip, name = ip_name.split(',')

    if list_type == 'block':
        record = {
            'DST_IP': ip,
            'TLS_SNI': name,
        }
        if ip_address(record['DST_IP']).version == 4:
            ip4_que.put(record)
        else:
            ip6_que.put(record)
```

Code listing 4.5 Methods for Handling Expired Rules

4.2 IP-Based Detection & Filtration Module

The IP-based detection module uses only the destination IP address and TLS SNI value from the input flow to make decisions. The module hosts two sets, the blocklist and the allowlist, which are used byt the list-based detection. The module's algorithm is illustrated in Listing 4.6.

```
while True:
    flow = trap.receive()
    if (flow[DST_IP], flow[TLS_SNI]) in allowlist:
        continue
    if (flow[DST_IP], flow[TLS_SNI]) in blocklist:
        trap.send(merger_ifc, flow)
        continue
    # Filtration step
    if (flow[PACKETS] >= 120 and flow[PACKETS] != 0):
        trap.send(next_module_ifc, flow)
```

Code listing 4.6 IP Module Algorithm

The lists are regularly updated from the central Redis, which is done by the `RedisManager` class as shown in Listing 4.3. Another class called the IPDetectionModule, is in charge of periodically updating the lists and has a method to check whether a current IP-SNI pair is present in one of the lists. The class has a separate daemon thread for the periodic updates. The thread sleeps for a predefined interval and updates the list when awoken.

The IP-based module is the only one that uses multiple output interfaces. The first one is to send alerts to the NEMEA Merger if a DoH flow is detected, and the second one is for passing unknown records to the classifier module. If the flow record does not meet all filtration conditions, it is dropped and the module proceeds to the next record.

4.3 ML Classifier Module

The classifier module utilizes the ML model to classify flow records from the IP-based detection module. It has two threads. The main thread is responsible for receiving flow data from the input interface and for sending alerts if a flow is labeled as DoH. The second thread, named RecordWorker, is tasked with processing the flows with the ML model. Listing 4.7 depicts these two threads, and their purpose in the module.

```
records_que = Queue()
alerts_que = Queue()

# RecordWorker Thread
while True:
    batch = records_que.get_multiple()
    features = compute_features(batch)
    predictions = clf_model.predict(features)

    for i, prediction in predictions:
        if prediction == 1:
            alert_que.put(batch[i])

# Main Thread
while True:
    while not alerts_que.Empty():
        trap.send(next_module_ifc, alert_que.get())

    flow = trap.receive()
    records_que.put(flow)
```

Code listing 4.7 Classifier Module Algorithm

4.3.1 Multithreading

The main thread is indeed responsible for receiving and sending UniRec messages as well, since sending alerts from any other thread than the main thread caused big problems, typically resulting in a segmentation fault. In the early versions of this module, it was implemented to process only one flow at a time, meaning that every flow on the input was individually processed by the ML model. But by adding the other thread for this job and with the model's support of batch predictions, it can process multiple flow records through the ML model at a time. The computed flow features are stored in an array and are processed by the ML model in batch, which returns an array of predictions, indexed the same way as the passed feature array.

Both `records_que` and `alerts_que` are defined as Queue classes from the queue[7] Python

[7]https://docs.python.org/3/library/queue.html

library. This library implements thread-safe multi-producer, and multi-consumer queues, providing all the required locking semantics.

4.3.2 ML Hyperparameters

In Chapter 3 we talked about the hyperparameters tuning and the set of used hyperparameters for the final XGBoost classifier. In Listing 4.8 are the actual values, that were found with Optuna.

```
hyperparameters = {
    'eta': 0.05291182757260608,
    'gamma': 0.4695685061384431,
    'max_depth': 8,
    'subsample': 0.9428776019771409,
    'colsample_bytree': 0.7528176279468037,
    'lambda': 0.2509168917726876,
    'alpha': 0.9598152571784272,
    'n_estimators': 587,
    'tree_method': 'exact'
}
```

Code listing 4.8 XGBoost Hyperparameters Values

4.4 Active Verification & Rule Extraction Module

The last module is accountable for verifying that a DoH resolving service is actually running on the flow's destination IP address. The ML classifier has labeled the flow as DoH, but we need to verify that it is really capable of processing DoH queries and returning a DoH response.

4.4.1 Active Verification

We had to choose to verify whether a certain host provides DoH resolving or not. The first option was to use Nmap, short for Network Mapper, which is an open-source tool for network discovery and host monitoring, very popular among system and network administrators. The second option was to write our program to check for the DoH service. The program would take parameters like IP address and TLS SNI, connect to the destination, send different DoH requests, and check for the responses.

We have decided to use Nmap, primarily because of the optimized techniques and its efficiency. Furthermore, Nmap has numerous options with which it can be run, that can reduce the time needed to scan one host. To minimize the surface the individual scan has to cover, we specified to only scan port number 443, since HTTPS is typically running on that port. We also disabled the DNS resolution of the scanned host, as this can significantly speed up the entire verification process. The last option was to treat all hosts as online and skip the host discovery. Normally, Nmap first attempts to determine whether a host is online by sending a series of standard discovery packets, like a ping request or TCP SYN packets. By skipping this phase, Nmap immediately proceeds to scan the specified ports or services on the target host. The options can be seen on Listing 4.9.

```
nmap --script=dns-doh-check -n -Pn -p443 [IPs] --script-args=...
```

Code listing 4.9 Used Nmap Options

Another great advantage of Nmap is that it can scan multiple hosts at once. This is powerful and can significantly speed up the verification process, as you can gather multiple hosts and verify them in one Nmap call. Each Nmap call involves initialization processes, such as loading scripts or initializing the scanning engine, but also some finalization steps, such as processing results and cleaning up resources. When you scan multiple hosts with one command, this initialization and finalization overhead occurs only once. Moreover, Nmap is designed to perform scans efficiently by leveraging parallelism.

Nmap Script

We used Nmap with `dns-doh-check`[8] script written by Tomáš Čejka. The script was slightly modified to accept the TLS SNI parameter and use it when establishing a TLS connection. Because we used Nmap to scan multiple hosts at once, we had to come up with how to pass the TLS SNI argument. The argument is passed on in a formatted string, where each SNI parameter is keyed by the host IP address. Since Nmap takes the argument value and use it to all hosts, this was the only way to pass different argument values to different hosts. The only problem occurs if there are two hosts with the same IP address but different SNI values in one Nmap call. As the argument is keyed by IP addresses, Nmap would always use only one of the SNI parameters from the argument. Therefore, this situation is neglected, and none Nmap calls contain duplicated IP addresses.

Listing 4.10 shows the example of how the argument is formatted and the argument processing logic inside the Nmap script. Nmap scripts are written in the Lua[9] programming language, and the script is called on every scanned host. The `host.ip` represents the IP address of the current host, ensuring that the right SNI argument will be used.

```
    nmap --script-args='4.4.4.4.sni=example.com,1.1.1.1.sni=abc.cz'

local arg_key = host.ip .. ".sni"
local tls_sni = stdnse.get_script_args(arg_key)
```

Code listing 4.10 Nmap Argument Parsing Logic

The script performs 6 DoH checks—both HTTP versions and all three request methods (GET, POST, and JSON). An example of one of the checks is shown in Listing 4.11, and corresponds to an HTTP/1.1 JSON request. The `result` variable is initially set to false and is changed only if some of the tests succeed.

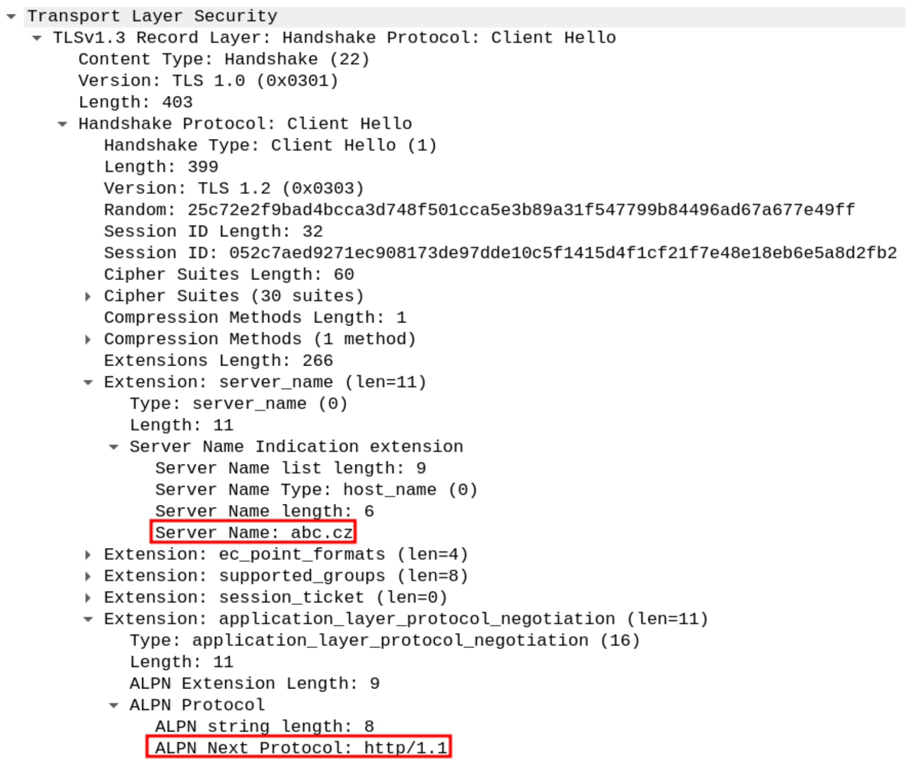
```
local query1 = '/dns-query?name=www.example.com&type=A'
local target = host.ip..":"..port.number

local req = http_request.new_from_uri("https://"..target..query1)
req.ctx = tlsctx
req.sendname = tls_sni
req.headers:append("accept", "application/dns-json")
req.headers:upsert("user-agent", "example/client")
req.version = 1.1

local headers, stream = req:go(timeout)
if headers and headers:get(":status") == "200" and headers:get("
    content-type") == "application/dns-json" then
    result = true
end
```

Code listing 4.11 Nmap Script DoH Check Example

To ensure the script is working properly, especially regarding the HTTP version and the used SNI value when establishing a TLS connection, we intercepted the communication with Wire-

[8]https://github.com/stratosphereips/DoH-Research/blob/main/nmap-script/dns-doh-check.nse
[9]https://www.lua.org/

shark[10]. We observed the TLS handshake packets and particularly inspected the SNI and Application Layer Protocol Negotiation (ALPN) values. The ALPN extension in the TLS handshake allows the client and server to negotiate which protocol to use over the secure TLS connection. This happens before the application layer protocol begins, ensuring that both parties have agreed on which protocol to use. Figure 4.2 is a screenshot from Wireshark of the Client Hello packet in the TLS handshake, and you can see that the SNI and also the ALPN version were set as intended.



```
▾ Transport Layer Security
  ▾ TLSv1.3 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 403
    ▾ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 399
        Version: TLS 1.2 (0x0303)
        Random: 25c72e2f9bad4bcca3d748f501cca5e3b89a31f547799b84496ad67a677e49ff
        Session ID Length: 32
        Session ID: 052c7aed9271ec908173de97dde10c5f1415d4f1cf21f7e48e18eb6e5a8d2fb2
        Cipher Suites Length: 60
      ▸ Cipher Suites (30 suites)
        Compression Methods Length: 1
      ▸ Compression Methods (1 method)
        Extensions Length: 266
      ▾ Extension: server_name (len=11)
          Type: server_name (0)
          Length: 11
        ▾ Server Name Indication extension
            Server Name list length: 9
            Server Name Type: host_name (0)
            Server Name length: 6
            Server Name: abc.cz
      ▸ Extension: ec_point_formats (len=4)
      ▸ Extension: supported_groups (len=8)
      ▸ Extension: session_ticket (len=0)
      ▾ Extension: application_layer_protocol_negotiation (len=11)
          Type: application_layer_protocol_negotiation (16)
          Length: 11
          ALPN Extension Length: 9
        ▾ ALPN Protocol
            ALPN string length: 8
            ALPN Next Protocol: http/1.1
```

Figure 4.2 TLS SNI and ALPN Wireshark Capture

```
Nmap scan report for 8.8.8.8
Host is up (0.0063s latency).

PORT     STATE SERVICE
443/tcp open  https
|_dns-doh-check: true
```

Code listing 4.12 Nmap Output Example

The output of a Nmap scan is shown in Listing 4.12. When multiple hosts are scanned, outputs like this are listed one after another. For processing and parsing the output, the re[11] Python library, providing regular expression (regex) matching, was utilized. Initially, the output was split by scanned IP addresses into individual reports. Then, we iterate through the reports and search for the `true` or `false` that the script has returned. Listing 4.13 demonstrates the logic of parsing the Nmap output.

[10]https://www.wireshark.org/
[11]https://docs.python.org/3/library/re.html

```
def parse_output(self, nmap_output):

# split the output into individual reports for each address
scan_report = re.split(r"Nmap scan report for", nmap_output)

# try to find the IP address and dns-doh-check pattern in every
    subreport
for report in scan_report:
    ip = self.ip_regex_compiled.search(report)
    doh_true = self.doh_true_compiled.search(report)
    doh_false = self.doh_false_compiled.search(report)

    # if both found, set value for the IP to True or False
    if ip and doh_true:
        self.scan_results[ip.group()] = True
    elif ip and doh_false:
        self.scan_results[ip.group()] = False
```

Code listing 4.13 Nmap Output Parsing Logic

4.4.2 IP-Rule Extraction

After the Nmap output is parsed and all scanned hosts are verified, the results are stored in the `scan_results` list, keyed by the IP address. The `send_alerts` method is called and is responsible for setting rules via the `RedisManager`, as well as for inserting DoH alerts into the alerts queue. The method is depicted in Listing 4.14.

```
def send_alerts(self, records):
    global alerts_que

    for record in records:
        ip = str(record['DST_IP'])
        name = record["TLS_SNI"]
        if self.scan_results[ip] == True:
            redis_manager.set_rule(ip, name, 'block')
            alerts_que.put(record)
        else:
            redis_manager.set_rule(ip, name, 'allow')
```

Code listing 4.14 Setting Rules to Rule Database

4.4.3 Multithreading

The idea of multithreading in this module is very similar to the classifier module. The main thread is responsible for receiving and sending UniRec messages, while other threads—called ConsumerThreads—are used to prepare the Nmap script argument, call Nmap, process its output, and extract rules for the database. The algorithm of this module is illustrated in Listing 4.15.

```
ip4_que = Queue()
ip6_que = Queue()
alerts_que = Queue()

# ConsumerThreads
class ConsumerThread(Thread):

    scan_results = []
    ip_que = ip4_que or ip6_que

    def send_alerts(records):
        for record in records:
            if scan_results[record[DST_IP]] == True:
                set_block_rule(record[DST_IP], record[TLS_SNI])
                alerts_que.put(record)
            else:
                set_allow_rule(record[DST_IP], record[TLS_SNI])

    while True:
        records = ip_que.get_multiple()
        targets = []
        for record in records:
            scan_results[record[DST_IP]] = False
            targets.append((record[DST_IP], record[TLS_SNI]))

        parse_output(run_nmap(targets))
        send_alerts()


# Main Thread
while True:
    while not alerts_que.Empty():
        trap.send(next_module_ifc, alert_que.get())

    flow = trap.receive()
    if record[DST_IP].isIpv4:
        ip4_que.put(flow)
    else:
        ip6_que.put(flow)
```

Code listing 4.15 Active Verification Module Algorithm

There are separate threads for IPv4 and IPv6 addresses because you cannot call Nmap with both address types at once. The whole flow records are stored in a thread-safe Queue object, from where they are taken and processed by ConsumerThreads. The user can specify how many threads there will be for each Queue, thus determining how many threads will process IPv4 or IPv6 addresses.

Scan results for a single host can be either `True` or `False`. The initial value is `False` and changes only when the Nmap scan results in `True`. There are situations where the scanned host is behind a firewall that filters and drops the scan packets. In such cases, the scan result value remains `False`, and an allow rule will be extracted. This is because typically, the host will remain behind the firewall, and the situation will not change. We do not want to repeatedly check and attempt to verify the host since it would be extremely intrusive. Therefore, the allow rule is inserted into the database, ensuring that the host will be checked again once the rule expires.

4.5 Supplementary Files

Configuration

Since many detector parameters can differ based on the platform it runs on, we have agreed to have a single configuration file for easier management. It has three sections—`redis`, `nmap`, and `clf`. In the `redis` section, the user can define which Redis database to connect to, including the host and port where it runs, as well as the validity of individual rules inside the database and the block/allow lists. `Nmap` section allows you to configure the multithreading options in the verification module since this depends on the available resources. The `clf` section is to set up the classification module, its ML model file, and also how many RecordWorker threads there should be.

ML Model

After an ML model is trained and hyperparameters tuned, it has to be exported so it can be then used in other processes. XGBoost library provides methods for saving and loading the XGBoost models in JSON format. The model is saved with a method called `save_model` and then can be loaded with `load_model`.

The ML classifier module requires this file and will not run without its presence. You can use practically any ML model that supports batch predictions (and was trained on the same feature set) and declare its location in the configuration file. The only condition is that it has to be a JSON file exported with the XGBoost library.

Logs

Each module has its own log file, to continuously report its work or to inform about errors. This is implemented with logging[12] Python module, which provides flexible event logging.

[12]https://docs.python.org/3/library/logging.html

Chapter 5
Evaluation

This chapter describes the methods used to evaluate the implemented detector from the previous Section. The ML model is evaluated based on the common metrics for evaluating any ML classifier. The detector's modules were individually tested for their correct functioning and error handling. Finally, the whole detector performance was tested on the dataset presented in Chapter 2, to measure its resource requirements.

5.1 ML Performance

Evaluating the performance of an ML model involves examining how well the model can perform on unseen data. Standard metrics for this are typically based on the numbers of true and false positives and negatives, which stand for:

- True Positives (TP): Correct predictions of positive class
- False Positives (FP): Incorrect predictions of positive class
- True Negatives (TN): Correct predictions of negative class
- False Negatives (FN): Incorrect predictions of negative class

A confusion matrix summarizes the performance of an ML model while displaying the numbers of actual and predicted labels, thus the TP, FP, TN, and FN values. The confusion matrix of our XGBoost classifier can be seen in Figure 5.1.

The accuracy metric represents the ratio of correct predictions and is calculated as the sum of correct predictions divided by the total number of predictions. This can be reliable only if we have a balanced dataset, meaning an equal number of samples in each class. However real-world datasets are usually heavily unbalanced, making this metric impractical.

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.1}$$

Precision and recall are other metrics commonly used when evaluating an ML model. Precision is a metric measuring how often our model correctly predicts a positive class (DoH flow in our case). It answers the question: how often the DoH predictions are correct? Recall is a metric that measures how often our model correctly predicts a positive class from all the actual positive samples in the dataset. It answers the question: can the model find all instances of DoH flows?

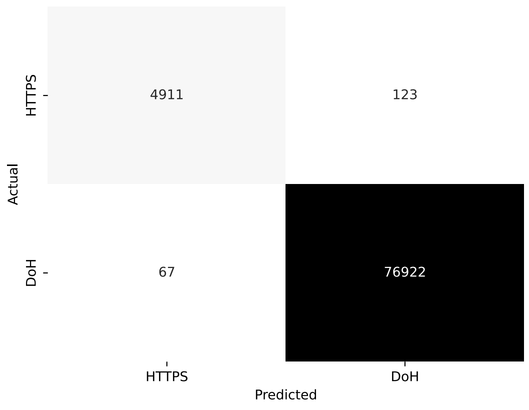


Figure 5.1 Consfusion Matrix od XGBoost Classifier

$$precision = \frac{TP}{TP + FP} \tag{5.2}$$

$$recall = \frac{TP}{TP + FN} \tag{5.3}$$

F1 score is another metric that combines these two and is computed as a harmonic mean of the precision and recall scores. The harmonic mean braces similar values for precision and recall, therefore, the more they deviate from each other, the worse the harmonic mean. F1 score can be computed as:

$$F1 = \frac{2 * precision * recall}{precision + recall} \tag{5.4}$$

The dataset was initially filtered of flows not having at least 120 packets, as described in Section 3.3, and randomly split into 70-30 training and testing parts, where the testing part was used for the final evaluation of the model. It was split with scikit-learn [1] `train_test_split` function, with the `random_state` value set to 422. The confusion matrix is depicted in Figure 5.1 and the individual metric scores can be seen in Table 5.1.

Precision	Recall	F1 score
0.9983	0.9991	0.9987

Table 5.1 Evaluation Metric Scores of the XGBoost Classifier

5.2 Correctness Testing

The testing methodology for the right functioning was iterative in nature, consisting of individual module tests, and progressing to inter-module communication. Modules were continuously tested during the implementation phase to validate its internal operations. Every program-failing operation is encapsulated with a try-catch block to handle possible exceptions, logging the event, but preventing the program from unwanted terminating.

[1]https://scikit-learn.org/stable/

5.3 Whole Detector Performance

The evaluation dataset contains 366,385 flow records, but the active verification was not performed since the dataset is over two years old. Instead, the ground truth labels were used. Nevertheless, the filtration step filtered most of the records and left 6057 DoH and 1910 non-DoH flows for further detection. The ML classifier then produced only 2 FN, meaning that two DoH flows were labeled as non-DoH and 3 FP which are expected to be eliminated by the active verification. The whole system thus achieved an F1 score of 0.9998 with precision of 1.0 and recall of 0.9997.

We also evaluated the detector performance in the manner of memory consumption and time process efficiency. The time was measured with the time [2] Python library accordingly: `start` variable was set at the beginning of the main loop of the module, and the `end` variable at the very end. The difference between those two timestamps represents the process time.

Two different methodologies were employed to measure the memory consumption: the htop [3] command-line utility and the psutil [4] Python library. Below is described the memory consumption and time process efficiency of each module.

IP-Based Detection & Filtration Module

The first module has to process every flow from the given dataset. The time was computed as an average of 6 independent measurements, resulting in 2.161 seconds needed to process more than 350 thousand records. This means it can process one flow under 6 microseconds.

Observations from htop indicated a per-process resident set size (RES) of approximately 64 MB. This measurement reflects the non-swapped physical memory that the process has used. In parallel, the psuitl reported consistent memory usage around 63 to 64 MB, which closely aligns with the htop measurements.

ML Classifier Module

The classifier module process flows that passed the filter in the previous module. From the given dataset, only 7967 flows passed the filter. Such amount of flows can be processed by this module in average of 1.348 seconds. This means that it can process one flow in 169 microseconds.

In the evaluation of the second NEMEA module, the top monitoring tool revealed that instance of the Python script allocates approximately 171 MB of physical memory (RES). The psutil tool reported a total memory consumption of around 175 MB.

Active Verification & Rule Extraction Module

Number of Threads	Total Time	Time per Flow
1	41.976 s	7.792 ms
2	20.1678 s	3.7437 ms
4	11.3937 s	2.115 ms

Table 5.2 Comparison of Process Time Efficiency with Different Thread Settings

Despite the obsolesce of the dataset and possible wrong results it produces, the active verification was performed just to evaluate the last module. Because this module is the most resource-consuming and heavily relies on multithreading, I wanted to test the performance of this module for various thread setting options. The previous ML module labeled 6093 flows as

[2]https://docs.python.org/3/library/time.html
[3]https://linux.die.net/man/1/htop
[4]https://psutil.readthedocs.io/en/latest/

DoH flows, where 5387 were IPv4 and the rest, 706, were IPv6. Since the testing platform did not support IPv6 scanning, I only verified the IPv4 flows. Table 5.2 compares the times needed to process all 5387 flows and the time needed to verify a single host.

The htop tool measured that an instance of this module allocates around 71 MB of physical memory. The psutil reported a total memory consumption of approximately 69 to 70 MB, which again aligns well with the figure reported by htop.

Conclusion

Given the increasing use of DoH, it is crucial to develop methods that can effectively identify such traffic within high-speed network environments. The primary objective of this thesis was to implement an algorithm for the automated detection of DoH traffic. Other detection methods and algorithms that were mentioned are not capable of reliable detection with minimized computational requirements.

The design of the detector in this thesis comprises several distinct phases, each tailored to address some aspect of the detection pipeline. Every phase contributes to enhancing the overall accuracy while also minimizing the computational resources, thus enabling effective detection within high-speed network environments. The shape and characteristics of DoH traffic were demonstrated on a series of graphs, which helps in understanding how the detection algorithm, especially the ML classifier, identifies DoH flows.

This bachelor thesis provides an implementation of a DoH detector within the NEMEA system framework, which utilizes different detection techniques to address the challenges of detecting encrypted DNS messages. The integration of the detector into the NEMEA framework not only showcases its practical applicability but also underscores its compatibility with existing network infrastructure and detection methodologies.

The effectiveness of the detector was evaluated on a real-world dataset, where it showed high accuracy and processing efficiency. The ML classifier was evaluated using common metrics for evaluating any ML model. The model showed exceptional results, even though it uses standard flow telemetry with only 4 features, which makes the model very simple (in comparison to other ML models for DoH classification).

Future work could employ other ML models with different feature sets aiming on other DoH distinctions. It is possible to merge outputs of more than one ML model, which could lead to more comprehensive detection results. By utilizing diverse ML approaches specializing in other DoH traffic characteristics, we can potentially enhance the detection accuracy and minimize the number of false positives generated by the ML module.

Bibliography

[1] Nevil Brownlee. *RTFM: Applicability Statement.* RFC 2721. Oct. 1999. DOI: 10.17487/RFC2721. URL: https://www.rfc-editor.org/info/rfc2721.

[2] Kimo Bumanglag and Houssain Kettani. “On the Impact of DNS Over HTTPS Paradigm on Cyber Systems”. In: *2020 3rd International Conference on Information and Computer Technologies (ICICT).* IEEE, Mar. 2020. DOI: 10.1109/icict50521.2020.00085. URL: http://dx.doi.org/10.1109/ICICT50521.2020.00085.

[3] Tomas Cejka et al. “NEMEA: A Framework for Network Traffic Analysis”. In: *12th International Conference on Network and Service Management (CNSM 2016).* 2016. DOI: 10.1109/CNSM.2016.7818417. URL: http://dx.doi.org/10.1109/CNSM.2016.7818417.

[4] Cisco Systems. *Cisco Annual Security Report.* Online. 2016. URL: https://mkto.cisco.com/rs/564-whv-323/images/cisco-asr-2016.pdf.

[5] Cisco Systems, Inc. *NetFlow Export Datagram Format.* [online], Accessed on 04-05-2024. Sept. 2007. URL: https://www.cisco.com/c/en/us/td/docs/net_mgmt/netflow_collection_engine/3-6/user/guide/format.html.

[6] B. Claise. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information.* Tech. rep. Sept. 2013. URL: https://datatracker.ietf.org/doc/html/rfc7011.

[7] B. Claise. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information.* Tech. rep. Jan. 2008. URL: https://datatracker.ietf.org/doc/html/rfc5101.

[8] Benoît Claise. *Cisco Systems NetFlow Services Export Version 9.* RFC 3954. Oct. 2004. DOI: 10.17487/RFC3954. URL: https://www.rfc-editor.org/info/rfc3954.

[9] Hannes Federrath et al. “Privacy-Preserving DNS: Analysis of Broadcast, Range Queries and Mix-Based Protection Methods”. In: *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 2011, pp. 665–683. ISBN: 9783642238222. DOI: 10.1007/978-3-642-23822-2_36. URL: http://dx.doi.org/10.1007/978-3-642-23822-2_36.

[10] Jan Fesl, Michal Konopa, and Jiří Jelínek. “A novel deep-learning based approach to DNS over HTTPS network traffic detection”. In: *International Journal of Electrical and Computer Engineering (IJECE)* 13.6 (Dec. 2023), p. 6691. ISSN: 2088-8708. DOI: 10.11591/ijece.v13i6.pp6691-6700. URL: http://dx.doi.org/10.11591/ijece.v13i6.pp6691-6700.

[11] Sebastián García et al. “Large Scale Analysis of DoH Deployment on the Internet”. In: *Lecture Notes in Computer Science.* Springer Nature Switzerland, 2022, pp. 145–165. ISBN: 9783031171437. DOI: 10.1007/978-3-031-17143-7_8. URL: http://dx.doi.org/10.1007/978-3-031-17143-7_8.

[12] Paul E. Hoffman and Patrick McManus. *DNS Queries over HTTPS (DoH).* Tech. rep. Internet Engineering Task Force (IETF), Oct. 2018. URL: https://tools.ietf.org/html/rfc8484.

[13] Rick Hofstede et al. "Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX". In: *IEEE Communications Surveys &amp; Tutorials* 16.4 (2014), pp. 2037–2064. ISSN: 1553-877X. DOI: 10.1109/comst.2014.2321898. URL: http://dx.doi.org/10.1109/COMST.2014.2321898.

[14] Karel Hynek et al. "Summary of DNS Over HTTPS Abuse". In: *IEEE Access* 10 (2022), pp. 54668–54680. ISSN: 2169-3536. DOI: 10.1109/access.2022.3175497. URL: http://dx.doi.org/10.1109/ACCESS.2022.3175497.

[15] Kamil Jarebek, Ondrej Rysavy, and Ivana Burgetova. "Measurement and Characterization of DNS over HTTPS Traffic". In: *arXiv* cs.NI.arXiv:2204.03975 (2022). arXiv:2204.03975v1. DOI: 10.48550/arXiv.2204.03975. URL: https://arxiv.org/abs/2204.03975.

[16] Kamil Jerabek et al. "DNS over HTTPS detection using standard flow telemetry". In: *IEEE Access* 11 (2023), pp. 50000–50012. DOI: 10.1109/access.2023.3275744.

[17] Kamil Jeřábek et al. "Collection of datasets with DNS over HTTPS traffic". In: *Data in Brief* 42 (June 2022), p. 108310. ISSN: 2352-3409. DOI: 10.1016/j.dib.2022.108310. URL: http://dx.doi.org/10.1016/j.dib.2022.108310.

[18] Yaser M. Banadaki. "Detecting Malicious DNS over HTTPS Traffic in Domain Name System using Machine Learning Classifiers". In: *Journal of Computer Sciences and Applications* 8.2 (Aug. 2020), pp. 46–55. ISSN: 2328-7268. DOI: 10.12691/jcsa-8-2-2. URL: http://dx.doi.org/10.12691/jcsa-8-2-2.

[19] P. Mockapetris. *Domain Names - Implementation and Specification.* Tech. rep. Nov. 1987. URL: https://datatracker.ietf.org/doc/html/rfc1035.

[20] Mohammadreza MontazeriShatoori et al. "Detection of DoH Tunnels using Time-series Classification of Encrypted Traffic". In: *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech).* IEEE, Aug. 2020. DOI: 10.1109/dasc-picom-cbdcom-cyberscitech49142.2020.00026. URL: http://dx.doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00026.

[21] J. Quittek. *Requirements for IP Flow Information Export (IPFIX).* Tech. rep. Oct. 2004. URL: https://datatracker.ietf.org/doc/html/rfc3917.

[22] G. Sadasivan. *Architecture for IP Flow Information Export.* Tech. rep. Mar. 2009. URL: https://datatracker.ietf.org/doc/html/rfc5470.

[23] *Using JSON · Cloudflare 1.1.1.1 docs — developers.cloudflare.com.* Online. Accessed 06-04-2024. URL: https://developers.cloudflare.com/1.1.1.1/encryption/dns-over-https/make-api-requests/dns-json.

[24] Dmitrii Vekshin, Karel Hynek, and Tomas Cejka. "DoH Insight: detecting DNS over HTTPS by machine learning". In: *Proceedings of the 15th International Conference on Availability, Reliability and Security.* ARES 2020. ACM, Aug. 2020. DOI: 10.1145/3407023.3409192. URL: http://dx.doi.org/10.1145/3407023.3409192.

Contents of the attachment

```
DoH_detector ............................................ DoH detector project
    config.ini ...................................... configuration file of modules
    dns-doh-check.nse ........................... Nmap script for host verification
    src
        nemea_IPlist_module.py ......................... IP-based detection module
        nemea_clf_module.py .................................. ML classifier module
        nemea_nmap_module.py ............................. DoH verification module
    model.json ................................................. exported ML model
text
    thesis.pdf ..................................................... thesis in PDF
    thesis_source_codes.zip ........................ compressed thesis source code
```