FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Analysis and detection of WireGuard traffic
Student:	Pavel Valach
Supervisor:	Ing. Tomáš Čejka, Ph.D.
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2023/2024

Instructions

Study the WireGuard protocol specification and flow-based network traffic monitoring and analysis, e.g., using an open-source ipfixprobe flow exporter [1].
Create a testing environment for experiments with WireGuard network traffic and its analysis.
Create a traffic dataset containing various types of traffic within WireGuard tunnels.
Design a detection algorithm based on deep packet inspection and a detection algorithm based on IP flow behavior.
Implement the algorithms as software prototypes (either in C or Python).
Design and implement a plugin for ipfixprobe to identify WireGuard connections.
Evaluate the developed software and describe the results of their throughput and precision.

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

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 4 January 2023 in Prague.

Bachelor's thesis

ANALYSIS AND DETECTION OF WIREGUARD TRAFFIC

Pavel Valach

Faculty of Information Technology
Department of Computer Systems
Supervisor: Ing. Tomáš Čejka Ph.D.
January 11, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Valach Pavel. *Analysis and detection of WireGuard traffic.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my thesis supervisor, Tomáš Čejka, whose patience with me I will always admire; then Karel Hynek for his valuable time spent on consultations, as well as Filip Němec and Václav Bartoš for their insights that helped to direct my work, and in general the entire network monitoring team at FIT and colleagues at CESNET for their support. And last but not least, to my family who never gave up on me either.
Ten years is a long time to study anything. I hope it was worth it. If nothing else, it is a nice, round number.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on January 11, 2024

Abstract

The monitoring of Internet traffic is becoming a necessity due to the requirements of today's world. In my work, I analyze the WireGuard protocol to develop an algorithm to detect it in network traffic. Moverover, the further goal of this work is to detect the category of the traffic passed inside the encrypted tunnel, without knowing the inner contents. In my work, I find that I can detect the presence of WireGuard from the packet data and create a detector for the ipfixprobe flow collector, which is part of the NEMEA framework for network traffic analysis. However, deep packet inspection requires the traffic content to be parsed and is insufficient to reveal the type of traffic contained within. That is where machine learning (ML) comes in. I collected seven (7) categories of data, both in cleartext and encapsulated in the WireGuard protocol. Then, I used several different ML classification algorithms, specifically AdaBoost and LightGBM, to train a decision tree that forms the basis of my models. They are trained to detect both whether the traffic is WireGuard or not and to detect the type of traffic (such as VoIP or web browsing). The result of my work is a functional processing plugin for ipfixprobe, the parameters of machine-learned models trained to detect WireGuard and various classes of traffic from IP flow characteristics, an evaluation of the throughput and precision of the software, and also a traffic dataset.

Keywords WireGuard, detection module, NEMEA, detection framework, ipfixprobe, detector plugin, machine learning

Abstrakt

Monitorování internetového provozu se vzhledem k požadavkům dnešního světa stává nutností. Ve své práci analyzuji protokol WireGuard, který se pokouším detekovat v síťovém provozu. Následně detekuji kategorii provozu procházejícího uvnitř šifrovaného tunelu, aniž bych měl přístup k dešifrovanému obsahu. Zjišťuji zde, že jsem schopen detekovat přítomnost protokolu WireGuard ze zachycených paketů (neboli provádím Deep Packet Inspection – DPI), a vytvářím detektor pro exportér toků ipfixprobe, který je součástí frameworku NEMEA pro analýzu síťového provozu. Nicméně, DPI vyžaduje přístup k obsahu síťového provozu, a také nedostačuje ke zjištění, jaký druh provozu tunelem prochází. Zde přichází ke slovu strojové učení. Posbíral jsem data pro sedm (7) kategorií provozu, a to jak v původní podobě, tak zapouzdřené v protokolu WireGuard. Následně jsem využil několika klasifikačních algoritmů, specificky AdaBoost a LightGBM, abych natrénoval rozhodovací strom, který pak posloužil jako základ mých modelů. Trénování proběhlo pro dva scénáře: 1) zda je provoz WireGuard či nikoliv, a 2) detekce kategorie provozu (jako např. VoIP či prohlížení webu). Výstupem práce je funkční detektor WireGuardu pro ipfixprobe, parametry natrénovaných modelů pro detekci WireGuardu a různých tříd provozu z charakteristik síťového provozu, zhodnocení přesnosti a výkonu detekce, a také sesbíraná datová sada.

Klíčová slova WireGuard, detekční modul, NEMEA, framework pro detekci, ipfixprobe, detekční plugin, strojové učení

Abbreviations

DH	Diffie-Hellman (key exchange algorithm)
DNS	Domain Name System
FIT CTU	Faculty of Information Technology, Czech Technical University in Prague
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
MitM	Man-in-the-Middle
MTU	Maximum transmission unit
OCSP	Online Certificate Status Protocol
P2P	Peer-to-Peer
SCP	Secure Copy Protocool
SIP	Session Initiation Protocol
SFTP	SSH File Transfer Protocol
SSH	Secure Shell (protocol)
VM	Virtual Machine
VPS	Virtual Private Server

Introduction

In today's network analysis, a great deal of attention is being put towards determining the nature of traffic. In that spirit, it is often useful to analyse not only the contents of individual packets, but also the context of the conversation – actors, protocols, duration, and other properties.

Flow analysis may be performed during the collection process, where it is possible to obtain significant detail on the traffic. The collected data are then distilled into a limited subset which is used for further analysis.

Today, a significant amount of network traffic is consisted of various tunnels and VPNs, often encrypted. WireGuard, with the protocol specification released in 2017, is a relatively new player in this field. In terms of performance, it already outperforms widely known IPsec and OpenVPN based solutions in various tests [1][2], and it has a lightweight code base [2]. Its author has developed a design that utilises properties of common computing architectures, allowing fast processing on common and modern hardware.

WireGuard has already gained some popularity among the security community and the general public. That popularity is growing and merits further research, as the usage of the protocol is likely to be higher in the near future. It is important to understand the protocol so that it can be identified and, if necessary, blocked in the network traffic.

Firstly, the thesis explains goals of this work. Then, there is an overview of the related previous works and explanation of the technologies used in this research and development. Next, the thesis "dives into" the implementation of the detectors, which are the main outcome of my work, including the strategic design choices. Finally, the developed software was evaluated and the results are provided in the dedicated chapter.

Goals

This work is focused on a direct extension of the NEMEA (Network Measurements Analysis) system, i.e., development of specialized components targetted on WireGuard communication. The NEMEA system as a base building stone is further explained in Section 1.5.

The particular main goals of this work are as follows:

- to study the specification of the WireGuard protocol and study flow-based network traffic monitoring and analysis,
- to design and create a testing environment for experiments with WireGuard network traffic and its analysis,
- to create a traffic dataset containing various types of traffic within WireGuard tunnels.

- to design a detection algorithm based on deep packet inspection and a detection algorithm based on IP flow behaviour,
- to implement the algorithms in the form of software prototypes (using C and Python), and
- to evaluate software prototypes and describe the results of their throughput and precision.

Contributions

My work is an indirect continuation of "Softwarový modul pro rozpoznání VPN v síťovém provozu", a master's thesis by Čtrnáctý, [3]. Čtrnáctý has analysed the OpenVPN and Cisco AnyConnect protocols and developed detection modules for the ipfixprobe flow exporter. I endeavor to develop comparable module for the WireGuard protocol and test its efficiency and detection rate. It should increase visibility into network traffic and allow future researchers to study the protocol and its usage.

Another contribution is the novel dataset of non-WireGuard and WireGuard flows including collected packet captures. These shall be labeled by the categories of traffic captured, and whether the traffic is inside the tunnel or the outside WireGuard traffic. The inspiration for the categories was the ISCXVPN2016 dataset [4] which contains full data captures and flow exports of OpenVPN traffic, which contains several different categories of traffic. To my knowledge, this will also be one of the first datasets with WireGuard flows, along the paper by Naas; Fesl (2023) which contains datasets from multiple VPNs, including WireGuard [5]; and the bachelor's thesis of Sandquist; Ersson (2023), concentrating on website fingerprinting capabilities (in this case limited to watching Twitch.tv) [6].

Finally, with the datasets, my work reveals whether it's possible to use machine learning to detect WireGuard only by utilizing time-based histograms and statistics of the flows. If successful, this could alleviate the need to collect full data packets to detect WireGuard. This is also what Čtrnáctý did for Cisco AnyConnect and OpenVPN, and my work is a continuation of such efforts.

Last but not least, the ability to detect specific categories of traffic without knowing the payload would allow their classification — for Quality of Service management, or network security management.

Chapter 1

Background and related works

1.1 VPN

A Virtual Private Network (VPN) is defined in [7] as a *"service that offers secure, reliable connectivity over a shared public network infrastructure such as the Internet."* More specifically, it was described by [8] as *"a way to provide secure communication between members of a group through use of the public telecommunication infrastructure, maintaining privacy through the use of a tunneling protocol and security procedures."*

Technically, a VPN nowadays is a secure IP tunnel which encapsulates IP datagrams for transit through the Internet, using *"cryptographic techniques to provide robust security and privacy"* [9] — or, in other words, ensuring confidentiality, integrity and authenticity of data. VPN implementations often seek to *"emulate the characteristics of an IP-based private network"*. [8] To list some well-known examples of protocols and technologies that can be used to create (or connect to) a VPN: IPsec/IKE, L2TP, PPTP, OpenVPN, Cisco AnyConnect, WireGuard, ZeroTier, and others. They are usually built on top of common IP transport protocols such as TCP or UDP; UDP, as a connection-less protocol, is usually preferred for data transmission due to significantly better performance. [9] As a side note, IPsec uses IP ESP (Encapsulating Security Payload) protocol for data transport, defined in [10].

Given the technical advancements in the area, it should be said that it is no longer just an enterprise matter. It is now feasible for a moderately experienced user to run a VPN service or for an inexperienced user to connect to such a service, sometimes without even knowing they are using a VPN (for example: "1.1.1.1 + WARP" operated by Cloudflare[11]).

1.2 WireGuard protocol

The WireGuard protocol is, according to [12], *"a secure network tunnel, operating at layer 3, implemented as a kernel virtual network interface for Linux, which aims to replace both IPsec for most use cases, as well as popular user space and/or TLS-based solutions like OpenVPN."*

The article with the protocol specification [12] was published in 2017 by Jason A. Donenfeld. Among the presented advantages are simplicity of use, high speed, proper core utilization on multi-core systems, and others. It has well-defined states and, for the outside observer, the functioning of the VPN interface appears to be stateless.

"Pre-shared static keys – Curve25519 points – are used for mutual authentication in the style of OpenSSH." [12] The protocol itself does not solve the pre-shared static key exchange between client and server, which makes it simpler. It also does not diminish the cryptographic

properties of the protocol, as it uses *"a single round trip key exchange, based on NoiseIK"*, and *"ChaCha20Poly1305 authenticated-encryption for encapsulation of packets in UDP"*. [12]

The WireGuard protocol also does not explicitly follow the server-client model; rather, during the handshake, the parties involved are being called Initiator and Responder (Section *V. Protocol & Cryptography*, [12]). With this model, any of the involved parties can initiate the connection, provided that they can reach the other party.

A Linux kernel module implementation was made by the same author. It is coupled with userspace utility `wg`, which is used to control the `wireguard` interface type created by the kernel module.

Thanks to the simplicity of the protocol, multiple independent implementations have already been developed, such as BoringTun [13] or TunSafe (available at `https://tunsafe.com/` with source code on TunSafe GitHub).

The protocol has received formal proofs regarding the cryptography used (e.g. [14], [15], [16]). According to [17], it is not, by default, post-quantum secure; mitigations can be applied, such as additional layer of encryption (pre-shared key), or additional post-quantum handshake on top of WireGuard.

1.2.1 Transport

WireGuard is a protocol which runs over UDP transport protocol [12]. From the paper, it is not apparent which port WireGuard uses as the default[1]. Currently, if the listening port is 0 or not set, the Linux kernel module of WireGuard chooses a random UDP port from the range of registered, dynamic or ephemeral ports during interface start.[2]

TCP mode is explicitly not supported by WireGuard reference implementation due to *"the classically terrible network performance of tunneling TCP-over-TCP"* [17], but may be achieved by supplementing WireGuard with projects such as udptunnel and udp2raw, or by using other implementations of WireGuard, such as TunSafe [20].

1.2.2 Communication flow

The following is a direct quote from the protocol specification [12], section *V. Protocol & Cryptography*:

"As mentioned prior, in order to begin sending encrypted encapsulated packets, a 1-RTT key exchange handshake must first take place. The initiator sends a message to the responder, and the responder sends a message back to the initiator. After this handshake, the initiator may send encrypted messages using a shared pair of symmetric keys, one for sending and one for receiving, to the responder, and following the first encrypted message from initator to responder, the responder may begin to send encrypted messages to the initiator."

A typical WireGuard communication flow is shown in Figure 1.1.

1.2.3 Message types

The WireGuard protocol contains a total of four message types:

- The handshake initiation message (ID 0x01)
- The handshake response to the initiation message (ID 0x02)

[1] However, the reference implementation at some point used port 51820 as the default, which incremented with multiple interfaces [18]. This port is also mentioned in the Quick Start guide [19].

[2] `https://github.com/WireGuard/wireguard-linux/blob/d0c2d5cdb991b07f9fefb44ad16bcadc10dcb1ae/include/uapi/linux/wireguard.h#L84`

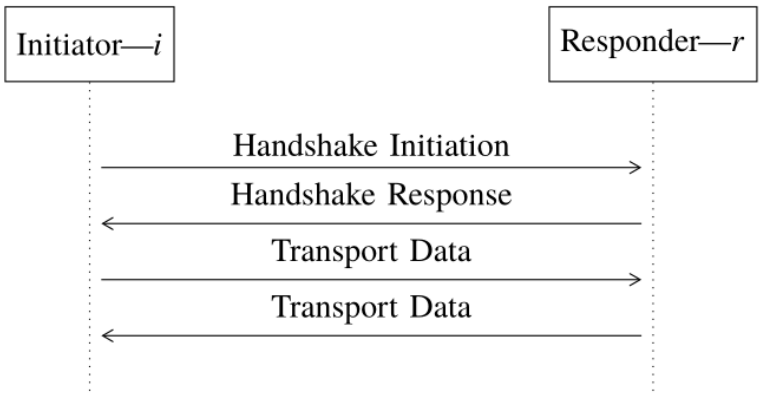

Figure 1.1 The typical communication flow. [12]

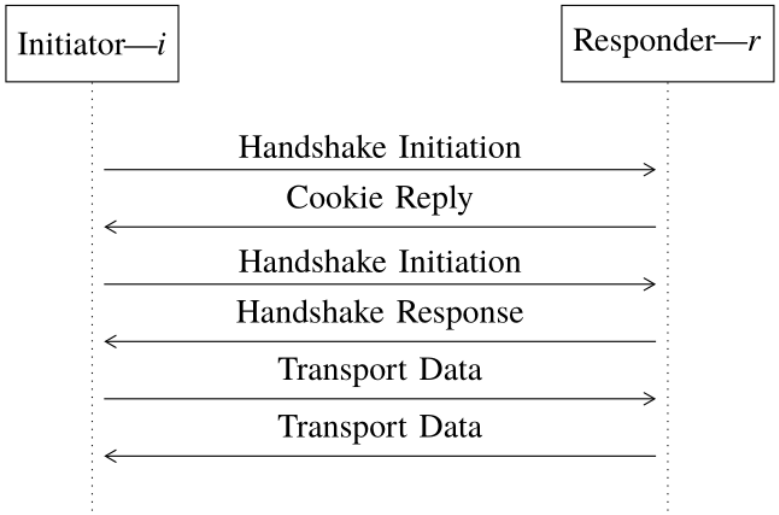

Figure 1.2 The communication flow with cookie messages. [12]

- The cookie challenge (ID 0x03)
- The encrypted session data (ID 0x04)

Each UDP payload starts with a message type (1 byte) and a reserved space (3 bytes). Some fields use authenticated encryption — AEAD, which adds additional 16 bytes to that field.

The first two message types are used during a handshake. The third type is sent by the peer when it receives a handshake message and is under load; the original handshake message is then discarded. The fourth is then used for data transfer, and also as a keep-alive packet.

Handshake Initiator to Responder (0x01)

The first handshake message (see Figure 1.3) contains these:

- **`sender index`** (4 bytes) — randomly generated, used for the remainder of the session by the responder. It contains no further information about the peer.

type = 1	reserved = 000 (3 bytes)
sender index (4 bytes)	
sender ephemeral public key (4 bytes)	
AEAD static public key (48 bytes)	
AEAD timestamp (28 bytes)	
`mac1` (16 bytes)	
`mac2` (16 bytes)	

Figure 1.3 The handshake initiation message [12]

- **`sender ephemeral public key`** (4 bytes) — a public key generated for the Diffie-Hellman (DH) key exchange algorithm.
- **`AEAD static public key`** (48 bytes) — a 32-byte encrypted and authenticated initiator public key.
- **`AEAD timestamp`** (28 bytes) — a 12-byte TAI64N timestamp [21], encrypted and authenticated.
- **`mac1`** and **`mac2`** — 16-byte Cookie MACs. `mac2` is only included in a reply to a cookie message.

Handshake Responder to Initiator (0x02)

The second handshake message (see Figure 1.4) contains these:

type = 2	reserved = 000 (3 bytes)
sender index (4 bytes)	receiver index (4 bytes)
sender ephemeral public key (4 bytes)	
AEAD empty portion (16 bytes)	
`mac1` (16 bytes)	
`mac2` (16 bytes)	

Figure 1.4 The handshake response message [12]

- **`sender index`** (4 bytes) — randomly generated, used for the remainder of the session by the responder. It contains no further information about the peer.
- **`receiver index`** (4 bytes) — the index of the receiving party, received in previous handshake step.
- **`sender ephemeral public key`** (4 bytes) — a public key generated for the Diffie-Hellman (DH) key exchange algorithm.
- **`AEAD empty portion`** (16 bytes) — encrypted and authenticated empty message.
- **`mac1`** and **`mac2`** — 16-byte Cookie MACs. `mac2` is only included in a reply to a cookie reply.

Cookie Reply (0x03)

This message is only sent as a response to a handshake when one of the peers is under high load. It looks like Figure 1.5 and the meaning of fields follows.

type = 3	reserved = 000 (3 bytes)
receiver index (4 bytes)	
nonce (24 bytes)	
AEAD cookie (32 bytes)	

Figure 1.5 The cookie challenge message – under load [12]

- `receiver index` (4 bytes) — the index of the receiving party, received in previous handshake step.
- `nonce` (24 bytes) — a value to prevent reusing of the cookie.
- `cookie` (32 bytes) — a 16-byte cookie which the peer uses, among other variables, to compute `mac2` in the handshake.

1.2.4 Data message

Citing [12], data messages are *"An encapsulated and encrypted IP packet that uses the secure session negotiated by the handshake."*

type = 4	reserved = 000 (3 bytes)
receiver index (4 bytes)	
counter (8 bytes)	
AEAD data (16-n bytes)	

Figure 1.6 The encrypted data message [12]

The message looks like Figure 1.6; the fields are:

- **receiver index** (4 bytes) — contains the random ID of the intended recipient.
- **counter** (8 bytes) — a monotonically increasing number indicating the order of packets.
- **AEAD data** (16-n bytes) — contains the actual data; it is encrypted using ChaCha20Poly1305 authenticated encryption, which adds 16 bytes to the length of the packet. The data payload itself is zero-byte-padded to the multiples of 16 bytes, mainly to make the life of an analyst a bit harder; the padding stops when the MTU of the WireGuard tunnel interface is reached. [12]

The data length of zero is used as a keepalive packet.

1.2.5 Constants and timers

As defined in [12], the state machine of WireGuard uses constants listed in Table 1.1.

Symbol	Value
REKEY-AFTER-MESSAGES	2^{60} messages
REJECT-AFTER-MESSAGES	$2^{64} - 2^{13} - 1$ messages
REKEY-AFTER-TIME	120 seconds
REJECT-AFTER-TIME	180 seconds
REKEY-ATTEMPT-TIME	90 seconds
REKEY-TIMEOUT	5 seconds
KEEPALIVE-TIMEOUT	10 seconds

Table 1.1 Constants of the WireGuard protocol

As the machine-learned models will use time-based characteristics of IP flows, these values and timers might aid detection, as the sequences of events are quite predictable.

If a peer has `persistent-keepalive` enabled for another peer, then it will transmit an empty data message towards the receiving peer each n seconds, where n is defined by the sending peer [12]. This serves mainly as a NAT-penetrating measure, keeping the connection open. This functionality is not enabled by default.

1.2.6 Handshake process

The first thing that the initiating side needs to do, is to send a handshake packet of type 1 to the responding peer. The initiator generates its own random index value, and sends their ephemeral public key, and their encrypted static public key, and initiates a DH key exchange using X25519 DH function. One of the properties sent by initiator is also an encrypted (and authenticated) timestamp, and the computed `mac1` value. The precise generation algorithms can be found in [12].

After the responder receives the packet, it checks whether the `mac1` is valid. If it is, then it repeats all computations, as the initiator did, only with the replaced operands of DH functions. If all checks pass, then the responder per replies with a type 2 message. It generates their own random index value, and sends both the sender index and receiver index back, along with their own ephemeral public key, authenticated empty data block, and a computed `mac1` value.

The initiator, after receiving the response, will check it the same as the responder has, and if the peer is valid. If that is true, then a key derivation function is run on both initiating and responding side; this key is then used for encrypting the traffic, and both peers are considered authenticated.

When the server is under load, the handshake follows a slightly different process, described briefly in Section 1.2.8.

1.2.7 Key management and rotation

As described in [12], *"WireGuard rests upon peers exchanging static public keys with each other* a priori*, as their static identities."* This means that the peers have to exchange their public keys by some other means.

The knowledge of each other's public keys is essential for authentication. During the handshake, random peer IDs are computed. These IDs are completely random, and are transmitted in each packet in clear; the only place where identity is actually linked to these IDs is during the handshake. If one has handshake packets, and if the peers' public keys are known to them, they can then confirm the identity of peers. [22]

For the purposes of my work, it is useful to know that the rotation of keys is performed every Rekey-After-Time seconds, or every Rekey-After-Messages messages, by using the same handshake process; the random indexes of the peers change, too. [12]

1.2.8 Protections against attacks

The WireGuard paper [12] describes the following protection mechanisms in detail:

Silence: The responder stays completely silent if an unauthenticated or improperly authenticated packet is received, and such packets will not influence its internal state. Thus it is invisible to network scanners and wrongly configured peers.

For a peer to respond to an initiation packet, the initiator must have the public key of the responder, and also the responder must have the public key of initiator in their list of peers. The exception is if the server is under load, in which case the server can reveal its existence with a cookie reply (see Denial-of-Service protection below).

Replay attack protection: If an attacker tries to do MitM attack and replay a captured packet from the initiator to the responder, then the responder should not respond again, if the timestamp is lower or equal to the previously received timestamp from the peer. This helps protect against replay attacks, which could disrupt the active session. However, if the state of the peer is lost for some reason (e.g. a reboot, or the interface has been recreated), then this can still generate a response from the server. [3] (I was able to simulate this behaviour while trying to induce a cookie reply.)

Denial-of-Service protection: WireGuard protocol contains integrated denial-of-service protection to prevent resource exhaustion, as computing Curve25519 signatures is CPU intensive. When a large number of packets is waiting in the receiving buffer to be processed, any handshake message (ID 0x01 or 0x02) with the correct `mac1`, but incorrect `mac2`, may trigger a special response, where a cookie reply is sent back to the initiator. The other party then has to compute a `mac2` signature from that cookie, keep that in their memory, and after Rekey-Timeout has passed, it shall send a new handshake packet, but now containing the `mac2` signature. Afterwards, the handshake and data session continues normally. [12]

1.2.9 Existing detection solutions

1.2.9.1 Wireshark dissector & decryptor

This is a Lua/C plugin for Wireshark created by Peter Wu in 2018–2019 [23]. It is now included in Wireshark releases. It uses stateless Deep Packet Inspection (DPI) to mark the packet as WireGuard and dissect each of the messages into parts, and it is also capable of confirming receiver identities of honest senders when their static public key is known. The decryptor is capable of decrypting the traffic, given that the handshake packets are captured and the private keys (static and ephemeral) are provided. [22]

1.2.9.2 R&S®PACE 2

This is an OEM protocol and application classification engine based on DPI technology, developed by ipoque GmbH. The company has announced WireGuard classification support in 2019. [24] I could find very little about the specific techniques used.

Other solutions are able of detecting (and maybe blocking) WireGuard using DPI, however specific information is hard to find.

[3] However, this is outside of the attack model assumed by [12], which assumes that if the responder's public key is already known, then the existence of the particular peer can be proven.

1.3 Deep Packet Inspection (DPI)

In literature, Deep Packet Inspection (DPI) is commonly referred to as a means of re-routing or filtering traffic based on its content. Although this method of traffic analysis takes considerable hardware resources, it has found widespread use in network security applications, because it is very flexible. It can, for example, be used to identify or drop traffic based on certain strings in the payload. [25]

In this work, I used DPI to extract payload from UDP packets, in order to make a determination whether they contained WireGuard protocol or not.

1.4 Flow-based network traffic detection and analysis

1.4.1 Flow definition

To better understand the topic of the work, an explanation of the term "flow" is in order.

In [26] a flow is defined as *"a set of IP packets passing an observation point in the network during a certain time interval, such that all packets belonging to a particular flow have a set of common properties"* These common properties *"may include packet header fields, such as source and destination IP addresses and port numbers, packet contents and meta-information."* [27]

To provide an example, we can have a set of packets from IP 1.2.3.4 to IP 5.6.7.8, using the source port 34567 and the destination port 443 (HTTPS), during a certain timeframe, which was started. That single connection would then form a flow with a direction identified usually by the first packet (e.g. who started the connection).

For TCP [28], a flow usually starts and ends with the corresponding connection. UDP [29] is a connection-less protocol; therefore, NetFlow protocol [30] considers that the flow has also ended when a certain time passes after the last packet between the two parties. Such a timer is also usually implemented for connection trackers of Network Address Translation (NAT) where the number of simultaneously active connections might be limited. [31]

1.4.2 Packet observation and flow exporting

Packet observation is how I gather data and export them into flows. In essence, I observe network traffic at a certain point in the infrastructure. [27] explains that *"Observation Points can be line cards or interfaces of packet forwarding devices, for example"*. Such traffic can then be sent to a flow exporter.

Flow metering and export stage, as described in [27], is *"where packets are aggregated into flows and flow records are exported"*. This can be done with a separate appliance, or a feature of a regular network switch or router, which usually have some form of flow export integrated in hardware. The main tasks of flow exporter, based on [27], *"are the reception, storage and preprocessing of flow data generated by the previous stage. Common pre-processing operations include aggregation, filtering, data compression, and summary generation."*

Flow exporters can also be programmed in software. [27] compares multiple software exporters, such as ipt-netflow, softflowd, nProbe, pmacct, QoF, Vermont, and YAF. Previously, I had used argus[4]. In this work, I used ipfixprobe (see Section 1.6), for which I wrote a plugin.

The exported flows are then transmitted to a flow collector. [27] lists, among others, these protocols for transmitting flow data: NetFlow [30], IP Flow Information eXport (IPFIX) [26], or sFlow (a sampled flow).

[4] https://openargus.org/

1.4.3 Flow collection

As described by [27], flow collectors *"receive, store, and pre-processs flow data from one or more flow exporters in the network"*. The type of storage varies for different needs. Volatile storage will usually be fast but smaller, such as RAM, and is used for short-lived data. Persistent storage would be bigger, usually slower, and used for larger flow collections or historical data. The latter is what I mostly used in this work.

To store flow data, one can use flat files, row-oriented databases, such as MySQL or PostgreSQL, and column-oriented databases. [27]

For the flat files, the usual formats are binary or text. Example of binary format is UniRec, further described in Section 1.5.1, which is what I used here for storing flows exported by ipfixprobe. Others include nfdump[5] or argus.

1.4.4 Flow analysis and its differences from packet analysis

The paper [27] explains principles of IP flow based monitoring in contrast to the packet based monitoring and deep packet inspection (DPI). DPI can be used to analyze individual packets and their payloads, usually collected by a packet dump. Flow analysis, on the other hand, observes exported flows, which constitute an aggregation of packets. Flow export usually does not contain packet payloads, only headers and general statistics such as session length. This helps with privacy issues regarding data collection. On the other hand, DPI can still provide insights into data and their structure, providing increased visibility into the network.

IP flow exports can be complemented by DPI when additional data from upper layers are needed. To give an example, a question from a DNS packet can appear in the IP flow export. This shows, citing [27], *how exporters with application awareness combine DPI with traditional flow export."*

1.5 NEMEA

NEMEA system (from "Network Measurement Analysis") is *"a stream-wise, flow-based and modular detection system for network traffic analysis. In practice, it is a set of independently running NEMEA modules that process continuously incoming data (messages)."* [32]

"Each module is an independent process which utilizes TRAP to communicate with other processes. Typically the modules are connected in a unidirectional tree where the input of the root is typically data on flows and the outputs of the leaves are the results of the analysis." [33]

One of the possible module trees is shown in figure 1.7.

The system is suitable for an on-the-fly analysis of the flow data (live or captured & stored). [33]

1.5.1 UniRec

UniRec (from "Unified Record") is a *"binary [data] format for storage and transfer of simple data records similar to plain C struct. In addition to the C struct it supports fields with variable length."* It is a generic data structure, where a particular format is given by *template*, i.e. a set of fields in a record. [33]

NEMEA uses UniRec as it has been designed to allow fast access to records and fields, and allows to define the template at run-time. This allows for dynamic extension of information elements during processing without interrupting the data flow. [33]

In my thesis, several UniRec fields are defined and added into ipfixprobe (Section 1.6) as a part of my WireGuard plugin.

[5] https://nfdump.sourceforge.net/

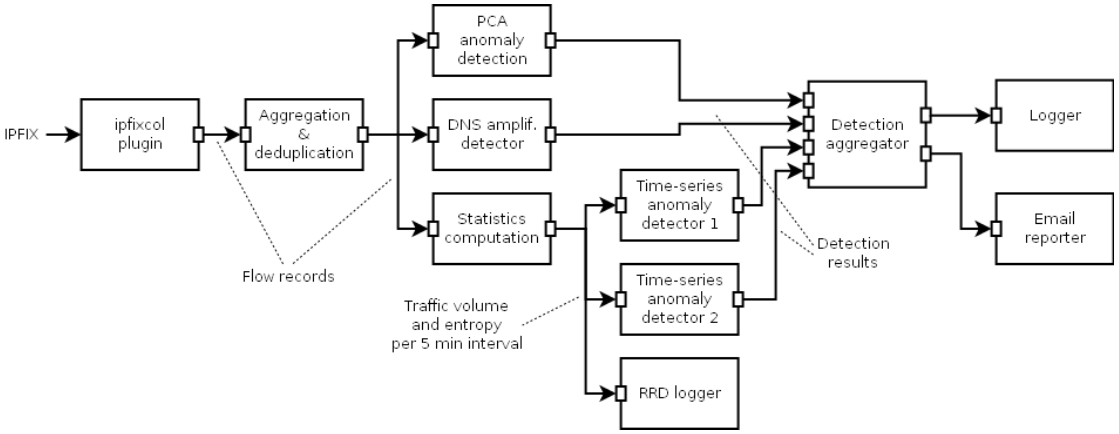

Figure 1.7 An example of a possible NEMEA processing pipeline (taken from [33]).

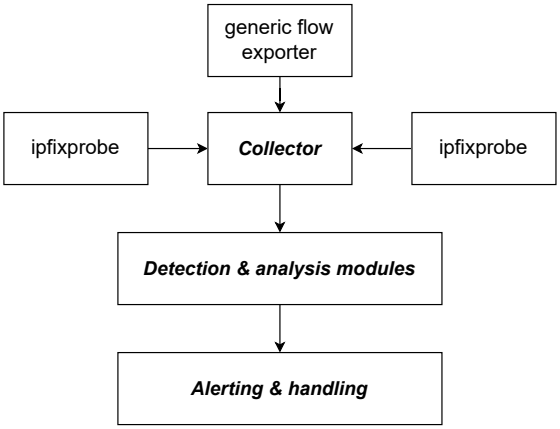

Figure 1.8 The high-level view of flow processing infrastructure.

1.5.2 Flow analysis in NEMEA

NEMEA is, by design, a *flow-based system* designed *"with respect to a stream-wise concept, i.e. data are analyzed continuously in memory with minimal data storage"* [33].

To achieve such goal, the processing of network flows is delegated to the *monitoring probes* which capture the traffic, convert it to *biflows* using ipfixprobe or a different *flow exporter*, and then send the flows to a *collector* for storage, and to NEMEA for analysis.

In NEMEA, the collected flows are subjected to analysis and detection modules, such as horizontal or vertical port scan detector, or DoS detector. *"The results of the analysis are statistics of traffic and alerts produced by various detection mechanisms."* [33]

The high-level view of a possible flow processing infrastructure can be seen in Figure 1.8.

1.5.3 VPN detection in NEMEA framework

The principal part of NEMEA framework, which matches protocols and detects the types of traffic flow, is ipfixprobe module and its exporters. I will be talking about it in detail in Section 1.6. As of March 2023, ipfixprobe includes a plugin for OpenVPN detection and also my WG (WireGuard) plugin which detects WireGuard protocol using DPI.

Several other protocols can be used to infer an existence of a VPN, such as DNS queries and

responses, which can be collected with the DNS plugin.

Most plugins export data using DPI. The question to answer is, whether there is a way to detect specific types of data flows inside VPN based on IP flow characteristics, rather than in the application layer (L7). In this thesis, I will attempt to present answers to some of these questions.

1.6 ipfixprobe

ipfixprobe is an IPFIX flow exporter. According to [34], it *"creates biflows from packet input and exports them to output interface"*.

It is capable of receiving packets via raw sockets, but also pcap, and NDP inputs from high-speed FPGA capture cards. The input information from the packets is then passed through plugins, which extract information from the packets into flows, mostly bi-directional (biflows). The content of these flows is determined by the plugins used.

ipfixprobe can output to several different interfaces, such as UniRec, IPFIX (RFC 5101), or the standard output for debugging. [34]

1.6.1 Plugins

ipfixprobe can be extended by new plugins for exporting various information from a flow. To date, many such plugins exist, e.g. for DNS, HTTP, NTP, SMTP, SIP, DNS-SD, and recently also OpenVPN or QUIC. Some of the plugins are collecting statistical information, for example PHISTS for the histograms of payload sizes and inter-packet times for each direction. [34] The plugins are mostly written in C++.

In this thesis, I am developing a plugin for detecting WireGuard traffic and exporting UniRec and IPFIX fields about such flows.

1.6.2 Creating a plugin

ipfixprobe has a documented process for adding new plugins [34], which involves using a script `process/create_plugin.sh` in the repository. This creates a source template called `<plugin>.cpp` and a header template `<plugin>.hpp` in the `process` directory.

Then, a programmer implements the following functions (or rather their subset). The descriptions come from the file `include/ipfixprobe/process.hpp` in the ipfixprobe repository.

- `pre_create` – called before a new flow is created.
- `post_create` – called after a new flow record is created.
- `pre_update` – called before an existing record is updated.
- `post_update` – called after an existing record is updated.
- `pre_export` – called before a flow record is exported from the cache.

The programmer also needs to define the UniRec fields, define IPFIX fields and add IPFIX template macro to the appropriate header file, and also write functions to fill the contents of IPFIX message and UniRec message.

1.7 Machine learning

When I work with data where I know or expect that they should possess some common property (or more), I want to teach the computer to recognise the patterns. This is where I want to use machine learning.

According to [35], *"Machine learning is a subfield of artificial intelligence (AI) concerned with algorithms that allow computers to learn. What this means, in most cases, is that an algorithm is given a set of data and infers information about the properties of the data—and that information allows it to make predictions about other data that it might see in the future."* This most common case, where the algorithm has the information, is also called supervised learning.

The paper [36] says that for supervised learning, *"The learner receives a set of labeled examples as training data and makes predictions for all unseen points. This is the most common scenario associated with classification, regression, and ranking problem."* This differs from unsupervised learning, where the set of examples is unlabeled. Other scenarios, like transductive inference, on-line learning, reinforcement learning and active learning are also described by [36].

All of my learning was supervised, as I became the entity supplying all the labeled training data.

As stated by [35], nowadays, many different machine-learning algorithms are available. They differ in capabilities and in the suitability for various problems. *"Some, such as decision trees, are transparent, so that an observer can totally understand the reasoning process undertaken by the machine. Others, such as neural networks, are blackbox, meaning that they produce an answer, but it's often very difficult to reproduce the reasoning behind it."*

Also according to [36], there are some kinds of problems commonly tackled with machine learning: text or document classification, natural language processing (NLP), speech processing, computer vision (such as optical character recognition — OCR), computational biology, fraud detecion, or network intrusion detection and information extraction systems.

1.7.1 Basic terminology

Some of the terms, as defined by [36], follow.

Classification is a *"problem of assigning a category to each item."* As an example, traffic flow can be classified into categories such as video calling, file download, or web browsing.

Example would be a particular item or instance of data used for learning or evaluation. Let's imagine a dataset of traffic flows. I will be using this term very loosely, though. If I mean a machine-learning example, I will clarify it as such.

Features are *"the set of attributes, often represented as a vector, associated to an example"*. This could be a length of a packet, or time between the arrival of packets.

Label is a value or category assigned to an example. Such as, I could say that a particular *example* of traffic is *file download*, and assign the *label* "file" to this traffic flow. Other, binary label, could be that the flow is WireGuard or not.

Hyperparameter is an input to the learning algorithm, usually specific to that algorithm. A set of them controls its inner workings.

Training sample is an *"example used to train a learning algorithm."* For my case, a training sample could be a set of traffic flows, appropriately labeled by a researcher (file, video chat and similar).

Validation sample is an *"example used to tune a learning algorithm."* A set of these is used to choose the appropriate hyperparameters.

Test sample is an *"example used to evaluate the performance of a learning algorithm. The test sample is separate from the training and validation data and is not made available in the learning stage."* For the traffic flows, I could pick a sample of traffic where I want the algorithm to predict the labels based on previous learning. Then the predictions can be compared with the true labels.

Loss function is *"a function that measures the difference, or loss, between a predicted label and a true label."* An example of such function is a binary / cross-entropy log loss.

1.7.2 Classification trees

As stated by [37], *"Classification and regression trees are machine-learning methods for constructing prediction models from data. The models are obtained by recursively partitioning the data space and fitting a simple prediction model within each partition. As a result, the partitioning can be represented graphically as a decision tree. Classification trees are designed for dependent variables that take a finite number of unordered values, with prediction error measured in terms of misclassification cost. Regression trees are for dependent variables that take continuous or ordered discrete values, with prediction error typically measured by the squared difference between the observed and predicted values."*

I will only use classification trees in my work, as all the values I am going to evaluate are discrete and their number is finite.

1.7.3 Reporting and scoring

As stated in the abstract of [38], *"Performance metrics in classification are fundamental in assessing the quality of learning methods and learned models."* However, there are many metrics which can be used, and it may be unclear which metric is to be used to measure accuracy of a particular classifier for a particular problem.

In general, [38] categorized performance metrics as follows.

- Threshold-based metrics with a *qualitative* understanding of error – such as *"accuracy, recall, mean F-measure (F-score)"* etc. According to [38], they are used *"when we want a model to minimise the number of errors"*.
- Metrics based on a *probabilistic* understanding of error – such as *"mean absolute error, mean squared error (Brier score), LogLoss (cross-entropy)"*, etc. They are *"useful when we want an assessment of the reliability of the classifiers, not only measuring when they fail but whether they have selected the wrong class with a high or low probability."*
- Metrics based on how well the model *ranks* the examples – which can then be used to *"to select the best instances of a set of data or when good class separation is crucial."*

For my use case, we both want to minimize the errors during a binary classification, and to assess the reliability of the trained classifiers. This would then imply that I would choose some of the threshold-based metrics, such as F-score, and then some of the probabilistic metrics, such as log loss.

Chapter 2
Approach

Since I need to detect a protocol, I will describe, what exact steps I need and what features I shall utilize for detection. But to do that, I need to create a dataset which would be later used for tests. I shall go into detail of capturing the necessary data, describing exactly how I did it and what architecture was used to perform the captures. Then, the detectors themselves had to be implemented. And, to be able to measure the precision and throughput, the analysis environment had to be created.

2.1 Creating data capture environment

I created my testing environment to aid with the preparation of experiments and with the collection of data for further analysis. Namely, the main objectives were:

- to have a functional and recent WireGuard server,
- to allow collection of data from the virtual machine, simultaneously the encrypted traffic and cleartext traffic, and
- to allow matching the *encrypted* WireGuard flows with the *cleartext* WireGuard flows.

I have decided to create my testing environment with a help of a server VM and client VM. The virtual machine allowed me to prepare a separate environment for testing and easily isolate the experiments from the rest of the network traffic.

2.1.1 Virtual machine – WireGuard server

The virtual machine was hosted at Sinkuleho dormitory in Prague, Dejvice, which is connected to the academic network provided by CTU in Prague and CESNET.

The virtual machine was running on an HP DL360 G7 server, on the Proxmox Virtual Environment 7.0-13 under the QEMU hypervisor. I set the emulated CPU type to *Host*, to improve performance. The VM had been assigned 1 GB of RAM and 2 vCPU cores, and 32 GB + 120 GB of mirrored SSD storage, running under ZFS.

A public IPv4 address was assigned and an IPv6 range was routed to the machine.

The WAN interface was protected by the firewalld firewall with a nftables backend, only allowing SSH, the WireGuard UDP port, and basic network protocols.

It also had no bandwidth limitations and was connected to a bridge with the available network bandwidth of 10 Gbps. However, it would not go higher than 1 Gbps, due to the client machines.

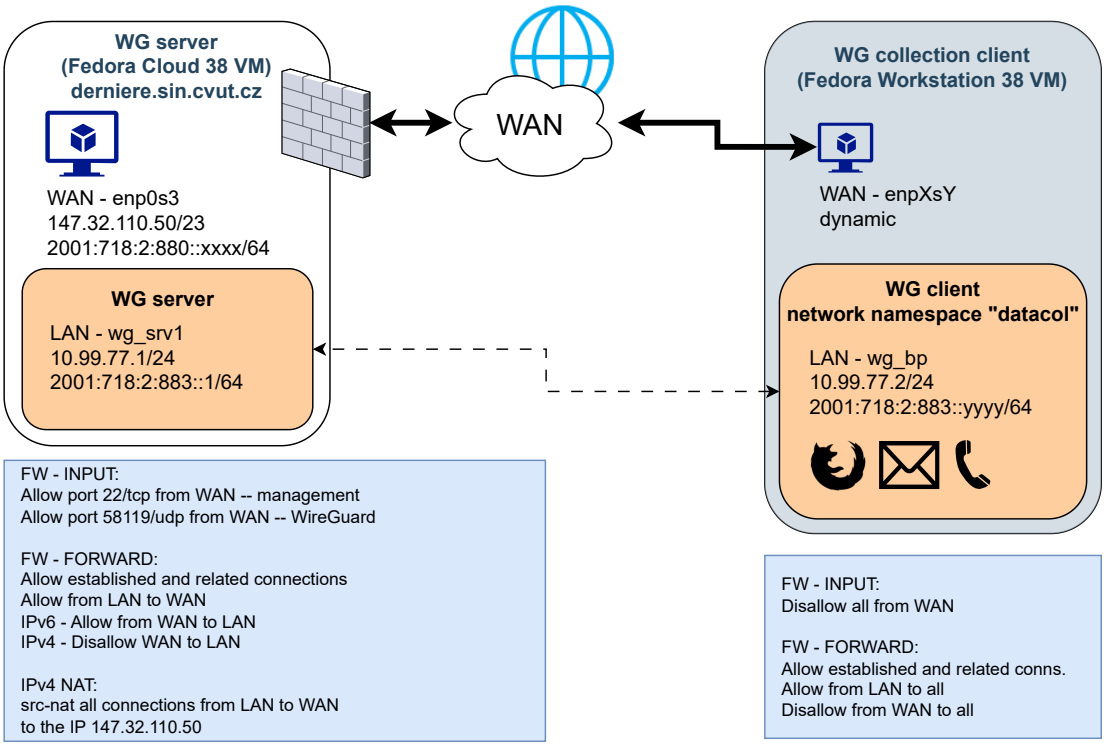

Figure 2.1 The high-level view of network architecture and firewall rules.

The OS is Fedora Cloud 38 amd64, with the latest system updates[1].
I called the virtual machine `derniera`, which means "the last performance" in Czech.

2.1.1.1 System configuration and software

I enabled *packet forwarding* for both IPv4 and IPv6, using the `sysctl` tool, which configures kernel parameters at runtime. The following two lines were placed in `/etc/sysctl.d/20-forwarding.conf`:

```
net.ipv4.ip_forward=1
net.ipv6.conf.all.forwarding=1
```

As for the software, I used OpenSSH to facilitate data and command transfer to and from the virtual machine. `tcpdump` and `tshark` were installed to collect raw packet data for debugging.

2.1.1.2 WireGuard setup

The current Linux kernel version of Fedora 38 Cloud has WireGuard already compiled, which is why only the userspace tool `wg` was needed to continue:

```
dnf install wireguard-tools
```

The preceding command installs not only the `wg` CLI tool, but also the `wg-quick` utility, which can be used to quickly set the interfaces up (or to tear them down) [39]. To automate

[1] As of June 2023.

the whole process, I utilized this tool with the following configuration file (note that the IPv6 interface ID is anonymized):

Code listing 2.1 WireGuard server configuration file

```
[Interface]
Address = 10.99.77.1/24
Address = 2001:718:2:883::1/64
ListenPort = 58119
PrivateKey = <private>

[Peer]
PublicKey = E75V2bG0sxQpXvzhEVq3QdNcQZwD74wuSak4BBQH8gY=
AllowedIPs = 10.99.77.2/32,
    2001:718:2:883:aaaa:bbbb:cccc:dddd/128
```

I stored the configuration at path `/etc/wireguard/wg_srv1.conf`, and then ran:

```
wg-quick up wg_srv1
```

To start up the WireGuard interface automatically after boot, it is then needed to use:

```
systemctl enable wg-quick@wg_srv1.service
```

2.1.1.3 `firewalld` configuration

I needed to open the port 58119/udp on the external (WAN) interface.

Because of the sole available IPv4 address and my intent to have more clients, I also performed Source NAT (masquerading) for the clients of the 10.99.77.0/24 prefix. Forwarding from internal to external zone was enabled in a special policy.

On the other hand, a whole prefix was assigned to IPv6, so I enabled forwarding for the whole IPv6 range.

I also forwarded a specific TCP and UDP port for the BitTorrent protocol to work.

For this, I used `firewalld` with the following configuration, which is divided into several categories, to:

- assign interfaces to the proper zones,
- enable receiving an IPv6 address,
- open the WireGuard port 58119/UDP,
- masquerade all trafic going from internal zone to external by using *SNAT* (that also implies the use of Connection Tracking, or conntrack),
- enable forwarding from internal to external zone (and, for IPv6, the other way around also - the client machines have a firewall).

The firewall configuration is listed in Code listing 2.2.

2.1.2 Virtual machine – WireGuard peer

The client was a virtual machine running Fedora Workstation 38 running in VirtualBox with 3 GB of RAM, 30 GB of disk space, and 128 GB of space for capture data. The VM had an emulated network interface (Intel PRO/1000 MT Desktop) with NAT and was connecting to the WireGuard VPN on server `derniera` created above.

Code listing 2.2 Server firewall configuration

```
  # move interfaces to right zones
  firewall-cmd --permanent --zone=external --add-interface=eth0
  firewall-cmd --permanent --zone=internal --add-interface=wg_srv1

  # get ipv6 to work
  firewall-cmd --permanent --zone=external --add-service=dhcpv6-client

  # enable wireguard UDP port
  firewall-cmd --permanent --zone=external --add-port=58119/udp

  # get masquerade and forwarding (incl. ipv6) to work
  firewall-cmd --permanent --zone=external --add-masquerade
  firewall-cmd --permanent --new-policy=wg_to_ext
  firewall-cmd --permanent --policy=wg_to_ext \
      --add-ingress-zone=internal
  firewall-cmd --permanent --policy=wg_to_ext \
      --add-egress-zone=external
  firewall-cmd --permanent --policy=wg_to_ext --set-priority=100
  firewall-cmd --permanent --policy=wg_to_ext --set-target=ACCEPT

  # forward traffic to 2001:718:2:883::/64, if possible
  firewall-cmd --permanent --zone=external \
      --add-source 2001:718:2:883::/64

  # IPv4 port forwarding for BitTorrent to work
  firewall-cmd --permanent --zone=external \
--add-forward-port=port=46942:proto=tcp:toport=46942:toaddr=10.99.77.2
  firewall-cmd --permanent --zone=external \
--add-forward-port=port=46942:proto=udp:toport=46942:toaddr=10.99.77.2

  firewall-cmd --reload
```

Code listing 2.3 Disabling segmentation and offload on the interfaces

```
ethtool -K enp0s3 tso off rx-gro-hw off \
                  gso off gro off \
                  tx-sctp-segmentation off
ethtool -K wg_bp tso off rx-gro-hw off \
                 gso off gro off \
                 tx-sctp-segmentation off
```

This machine captured both the cleartext and encrypted data. To collect data suitable for further processing by ipfixprobe, some system settings had to be modified.

2.1.2.1 System configuration and software

Linux network interfaces usually employ segmentation offload techniques to improve overall networking performance. Such techniques benefit from the combined capabilities of network interfaces, their firmware and drivers; however, they can also be performed in software. More details on segmentation offloading are described in [40].

To ensure that I capture the raw data without any segmentation being performed either by the OS or the NIC, I disabled the following features.

- TCP Segmentation Offload (TSO)
- Generic Segmentation Offload (GSO)
- Generic Receive Offload (GRO) – both in software and in hardware
- SCTP segmentation (`tx-sctp-segmentation`)

The commands for disabling all of the above are specified in Code listing 2.3.

2.1.2.2 Network configuration

The client was configured to minimize the noise in the captured data. First, `avahi-daemon` and similar services were turned off. Then, the network namespace `datacol` was created, and all the captured applications ran in that namespace. DNS name servers were set to public CESNET resolvers with IPv6 addresses 2001:718:1:1::2 and 2001:718:1:101::3.

The `systemd-resolved` service had to be turned off, as it interfered with the capture of DNS data in the namespace. That was done by disabling and masking the `systemd-resolved` service in `systemd`, then restarting NetworkManager. Then, the specific DNS servers were set in `/etc/netns/datacol/resolv.conf`, which served as the `/etc/resolv.conf` file while in the `datacol` namespace.

2.1.2.3 WireGuard configuration

The process of creating the WireGuard connection involved mlutiple steps. First, I enabled verbose mode in the kernel module `wireguard`. Then, I created the network namespace `datacol` and WireGuard interface called `wg_bp`, which I moved right after that to the `datacol` network namespace. By following this procedure, I ensured that the connection to the VPN would occur outside of the namespace, through the WAN, but all the communication inside the namespace would be routed through the VPN. The WireGuard local addresses were then added to the interface, the interface was brought up, and after that, the default route was added. Finally, as the last step, segmentation offloading was turned off on the `wg_bp` interface.

The `/etc/wireguard/wg_bp.conf` configuration file is displayed in Code listing 2.4.

Code listing 2.4 Peer WireGuard configuration file

```
[Interface]
ListenPort = 43400
PrivateKey = <private>

[Peer]
PublicKey = xI4tLOYTqFnCA361fAoRduxHelaaB1LXI6wZ9J92yFA=
Endpoint = derniere.sin.cvut.cz:58119
AllowedIPs = 0.0.0.0/0, ::/0
PersistentKeepalive = 25
```

Code listing 2.5 Peer WireGuard setup

```
echo module wireguard +p \
    > /sys/kernel/debug/dynamic_debug/control

ip netns add datacol
ip link add wg_bp type wireguard
ip link set wg_bp netns datacol
ip -n datacol addr add 10.99.77.2/24 dev wg_bp
ip -n datacol addr add \
    2001:718:2:883:aaaa:bbbb:cccc:dddd/64 dev wg_bp
ip netns exec datacol wg setconf \
    wg_bp /etc/wireguard/wg_bp.conf
ip -n datacol link set wg_bp up
ip -n datacol route add default dev wg_bp
ip -n datacol route add default \
    via 2001:718:2:883::1 dev wg_bp

ip netns exec datacol ethtool -K wg_bp \
    tso off rx-gro-hw off gso off gro off \
    tx-sctp-segmentation off
```

The shell script that ultimately performed the preparations, `enable_wg.sh`, can be seen in its entirety in Code listing 2.5.

This prepared the stage for captures, which were then accomplished using `tcpdump`.

For the purposes of quick review of the data, I also installed `wireshark`. This helped me quickly evaluate the data that I captured.

2.1.2.4 `firewalld` configuration

For the client, it was not necessary to modify the default firewall configuration. Output connections were enabled by default, and incoming traffic to the 46942/tcp and 46942/udp ports (or rather all the dynamic ports), which was later needed for BitTorrent, was implicitly allowed.

2.2 Creating the environment for analysis

First, I needed to evaluate the throughput and precision of my detectors. The DPI detector works with the raw packet captures, while the machine-learned model works on the flows and their statistical properties. But in both cases, I needed to extract flows from the captured traffic.

Code listing 2.6 Configuration options for nemea

```
./configure --with-pcap --with-nemea
```

Table 2.1 Versions of ipfixprobe and Nemea used

Name	Commit ID	Reference
ipfixprobe	af624f25eac2a7e2fbd22d3e30db97222a8402a6	heads/master
Nemea	195c6f8c0b2f201dbee178e7708369921b629838	heads/master
nemea-detectors	84f6f1b15f43f0b81e34c4859ff1ada6fb6e8270	heads/master
nemea-modules	c16766702209b4b899cf1ee101ae97c08e0e0b05	v2.1.0-1216-gc167667
nemea-framework	a554edfb3bb834218ba4a1268188a641cd07ae6f	v2.0.0-1247-ga554edf
nemea-supervisor	89afda171fc119db253235f7a8e0c71614296dec	v1.1.1-331-g89afda1

2.2.1 Flow extraction

I achieved this with my DPI processing plugin. For the use in statistical models, PSTATS, PHISTS and BSTATS plugins were also enabled. With this, I processed the data and output the necessary properties.

All of the plugins mentioned are now included in the default ipfixprobe configuration.

I needed to install ipfixprobe to convert the data captured into flows, and also NEMEA to use UniRec data format and underlying tools (`logger`) to convert data into the CSV format.

I compiled ipfixprobe from source code, along with NEMEA framework. This was necessary as the binary packages for Fedora 38 were not available at that time, and it also allowed for modifications of code and debugging.

The peer VM was useful to perform some basic preprocessing of the data.

The versions of software used are listed in Table 2.1.

From that point forward, I was able to export flows right after capture, including the necessary statistics.

2.2.2 Flow analysis and visualisation

I used the following libraries for analysis, designing and training the machine learning model:

- matplotlib 3.5.2
- numpy 1.22.0
- pandas 1.4.1
- scikit-learn 1.1.1
- optuna
- Feature Exploration Toolkit (nemea-fet)[2]
- LightGBM

I used an environment at my faculty to perform most of the analyses because of available resources. In December 2023, with the given set of packages — actually updated to the latest versions — I was able to perform the analysis on my laptop with enough RAM installed (at least 32 GB).

[2] https://docs.danieluhricek.cz/fet/

2.3 Dataset creation

Data for the dataset were collected during the months of August to December 2023. The process of collection was largely manual. This was done to ensure that the data represent, at least in some way, regular traffic.

The data was collected with two analyses in mind:

- whether the traffic is WireGuard or not,
- which category does the particular flow represent.

Traffic inside the tunnel (the cleartext) and the encrypted WireGuard tunnel traffic were captured in full and are a part of this work.

The cleartext inner traffic is Dataset A, and the encrypted WireGuard traffic is Dataset B.

2.3.1 Dataset labels

The data were labelled into several categories:

- **Web** — browsing the web using Firefox and Chrome
- **E-mail** — a regular work with e-mails (receiving and sending) with protocols IMAP and SMTP (over TLS)
- **Chat** — connecting to, and chatting via, IRC and Matrix protocols
- **Video** — playing short and long videos over YouTube
- **File download / upload** — SFTP, rsync over SSH, and HTTP(S)
- **P2P** — peer-to-peer BitTorrent traffic
- **Voice call / video call** — voice calls via SIP and voice/video calls via Skype

I took inspiration for the labels from the VPN-nonVPN dataset ISCXVPN2016 [4].

Some categories overlap with respect to the network protocols used (e.g., *web* and *video* or *P2P*), as several classes of traffic, which used to be different protocols, are now using web technologies underneath. However, I tried to make sure that the classes were distinct enough.

The exact setup for each category will be described later in this section.

2.3.2 Data collection and annotation methodology

Data capture was performed on the virtual machine set up as described in Section 2.1.2, primarily on networks connected to the CESNET national research and education network, such as the Czech Technical University in Prague and the National Technical Library in Prague. This resulted in low latency and high available bandwidth. To faithfully simulate the conditions of a regular user PC, I performed data captures on Wi-Fi networks as much as possible, with Wi-Fi 5 and 6 standards.

There are seven data labels: *web, email, chat, video, file, p2p, voice*. Each of them corresponds to a category of traffic that is being captured. Each capture has exactly one label. Both external (encrypted) and internal (cleartext) WireGuard traffic was captured in full to allow later analysis and customized conversion to flows.

Data capture was performed with `tcpdump` and saved in the PCAP files named as follows:
`capN-CATEGORY-DESCR/YYYY-MM-DD_in_wg.pcap`
and
`capN-CATEGORY-DESCR/YYYY-MM-DD_out_wan.pcap`
where:

Code listing 2.7 Capturing the inner WireGuard traffic

```
ip netns exec datacol \
    tcpdump -ni wg_bp -w "$DATADIR/‘date␣+%F‘-in_wg.pcap"
```

Code listing 2.8 Capturing the outer WireGuard traffic

```
tcpdump -ni enp0s3 -w "$DATADIR/‘date␣+%F‘-out_wan.pcap" \
    ’host␣147.32.110.50␣and␣udp␣port␣58119’
```

- `N` is the sequence number of capture,
- `CATEGORY` is the traffic label — one of *web, email, chat, video, file, p2p, voice*,
- `DESCR` is additional description of the data,
- `YYYY-MM-DD` is the date of capture,
- `in_wg` / `out_wan` means either inner (cleartext) or outer (encrypted) traffic.

The specific capture commands are specified in Code listing 2.7 and Code listing 2.8.

In addition, I also captured verbose kernel logs during the same period, using the command in Code listing 2.9.

Then, ipfixprobe was used to extract flows from the data capture files. Flows were capped to the maximum of 150 seconds, instead of 300. The command used is in Code listing 2.10.

2.3.2.1 Web

The web dataset was captured with Mozilla Firefox and Google Chrome. Firefox was running for approx. 2hrs and Chrome for approx. 2hrs 20min., without ad blocking.

The pattern of this experiment was browsing of the Czech and English Web, consisting of news sites, weather, sports, using Office 365 online editing apps, and a Mastodon instance, among others.

In Firefox, the standard tracking protection was enabled by default, and the setup was as close to default as possible. In Chrome, it was the same.

DNS over HTTPS was not explicitly enabled and, as the captured data shows, regular DNS queries were sent.

2.3.2.2 E-mail

The email dataset contains mostly IMAP and SMTP using Thunderbird 102.13.

I subscribed to the linux-kernel@vger.kernel.org mailing list to generate a steady flow of emails, which was ultimately much larger than expected.

There were two distinct attempts to capture emails, with a space of over two months in between. The second attempt involved downloading a large amount of emails being transmitted (approx. 90,000 new emails) after two months of inactivity. The total length of the captured data is 3 hours and 53 minutes.

Code listing 2.9 Capturing the kernel logs

```
journalctl -k -f --since=now \
    > "$DATADIR/‘date␣+%F‘-kernel"
```

Code listing 2.10 Extracting flows from data capture files

```
i="cap20-file-rsync-slow-download-100k-2/2023-12-11-out_wan.pcap"

ipfixprobe -i "pcap;file=$i" -p pstats -p phists -p bstats \
    -s "cache;a=150" \
    -o "unirec;i=f:${i}.trapcap:w;p=(pstats,phists,bstats)";
```

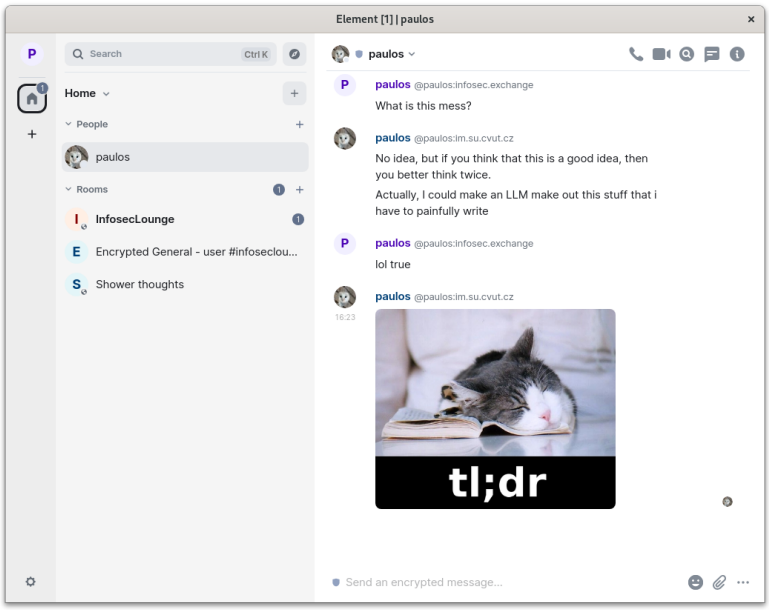

Figure 2.2 C(h)at capture

There is some noise generated by DNS requests and occasional HTTP requests by the Thunderbird client, the second of which is almost negligible.

2.3.2.3 Chat

The chat dataset contains IRC and Matrix.

For IRC, I used the Weechat client, connecting to three distinct IRC servers - libera.chat, hackint.net and rezosup.net, where there were 6 channels in total. In particular, #archlinux-offtopic on libera.chat was a high-traffic channel. I have also posted messages at irregular intervals. No direct data transfers (DCC) were attempted. The length of the first capture is around 52 minutes, and the second is around 1 hour and 40 minutes.

For Matrix, I used the Element client and my account at CTU Student Union homeserver, with 9 high-traffic channels with more than 500 members and around 20 of smaller ones. I joined #offtopic:archlinux.org to generate a large burst of traffic. Furthermore, I have tried to post text and rarely images at irregular intervals. The captured length is about 1 hour. Second Matrix capture involved a smaller account at a second homeserver, where there I was only joined to three channels, the largest of 160 members (161 with me), and one direct chat. For the last 20 minutes of capture, I also disabled read receipts and typing notifications on the capturing client, and also on my second client with which I generated some traffic. That capture took around 56 minutes.

Code listing 2.11 Commands used for HTTP downloads

```
wget -O /dev/null --limit-rate=100K \
    https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.4.8.tar.gz
wget -O /dev/null --limit-rate=10M \
    https://.../.../Fedora-Workstation-Live-x86_64-38-1.6.iso
```

Code listing 2.12 Commands used for SSH/SFTP downloads and uploads

```
# upload
scp ./linux-6.4.8.tar.gz anon_down@paulos.cz:
# download
scp anon_down@paulos.cz:linux-6.4.8.tar.gz /dev/null
```

2.3.2.4 Video

The video dataset was collected from YouTube. In total, 1 hour of video was played with advertisements, then 3 hours and 20 minutes of video without advertisements, both on YouTube.

There is some HTTP and DNS traffic around when the page was loading. This is a noise that would also be present in regular traffic, and therefore I felt it useful not to attempt to remove it.

2.3.2.5 File download / upload

This dataset was collected by downloading and uploading various files from various sources. It is probably the most reproducible part of this collection effort.

Among the downloads were:

- uloz.to: Zabbix Appliance 5.0.12 Netinstall ISO (capped by the server at approx. 300 KB/s)
- cdn.kernel.org: linux-6.4.8.tar.gz (capped in wget at 100 KB/s to simulate slow download speeds)
- download.fedoraproject.org: Fedora-Workstation-Live-x86_64-38-1.6.iso (capped in wget at 10 MB/s, then 100 KB/s)
- gitlab.freedesktop.org: NetworkManager-358c534e199c2cf4a72e4cbf26e83c8091f18629.tar (unpredictable speed variations, from 50 KB/s to 5 MB/s, total of 5 downloads)
- ixpeering.dl.sourceforge.net: systemrescue-10.02-amd64.iso (download from Australia, first without limitation, then limited at 50 KB/s for 30 minutes)

To bring some unpredictability into the mix, I have added FreeDesktop's Gitlab download into the mix, and two downloads from IX Australia were introduced for long Internet paths.

With SFTP/SCP downloads, two tests were made. First, through SFTP protocol only, was to download and upload a big file. The second, through protocols SFTP and rsync, was aimed at copying a large amount of rather small files (in the order of tens or hundreds of kBs). I performed the from the WireGuard peer to my personal VPS at VPSfree.cz, hosted in Prague.

For that purpose, I was copying only 'linux-6.4.8/drivers/media' with a total size of approx. 44 MB. The copying was from NVMe to a `tmpfs` filesystem (a "RAM disk") on my personal VPS server at VPSfree.cz. This was done to minimize possible I/O variances on the VPS side, where the disk is shared by a great number of people and the speed would have been unpredictably affected by this.

Server version was: `OpenSSH_8.4p1 Debian-5+deb11u1, OpenSSL 1.1.1n 15 Mar 2022`. Client version was `OpenSSH_9.0p1, OpenSSL 3.0.9 30 May 2023`.

Code listing 2.13 Commands used for small file transfer over SFTP and rsync

```
$ time scp -rq ./linux-6.4.8/drivers/media \
    anon_down@paulos.cz:/run/user/1003/linux-transfer-test-scp/
0.20user 0.63system 1:08.01elapsed 1%CPU (0avgtext+0avgdata
9708maxresident)k
0inputs+40outputs (0major+1871minor)pagefaults 0swaps

$ time rsync -a ./linux-6.4.8/drivers/media \
    anon_down@paulos.cz:/run/user/1003/linux-transfer-test/
0.05user 0.17system 0:03.66elapsed 6%CPU (0avgtext+0avgdata
14640maxresident)k
87792inputs+136outputs (4major+31714minor)pagefaults 0swaps
```

Fedora download took multiple attempts and there was a packet loss on the capture, until I limited the rate to 10 MB/s.

The data capture of the uloz.to transfer is not completely clean. At the beginning is a DNS query and at the very end is an OCSP query by Firefox to Google's servers.

2.3.2.6 P2P transfers (BitTorrent)

The Transmission client was used for BitTorrent transfers, on ports 46942/TCP and 46942/UDP. A combination of Debian 11 and 12 ISOs was used, as well as the Tears of Steel Blender movie[3], and ImageNet LSVRC 2012 Validation Set[4]. After the download was completed, I usually kept the downloaded files around for seeding.

Heavy unrestricted peer-to-peer traffic was present, with some web seed downloads, especially when Debian downloads were concerned. Some uploads were also made by the client; however, the total upload speed by the client rarely approached 1 MB/s and was mostly in the area of tens, maybe hundreds of KB/s, but mostly idle (discounting the state exchanges between the peers). In contrast, download speeds usually exceeded 5 MB/s, sometimes even 10 MB/s.

Since I was unable to disable web seeds in the client, I removed them using a Torrent File Editor[5], then loaded that into the BitTorrent client. Another approach would have been to disable HTTPS traffic completely.

2.3.2.7 Voice call / video call (labelled as voice)

I made voice calls over the SIP/RTP protocol and audio (or audio and video) calls using Microsoft Skype / Teams.

For SIP, a Linphone 4 client was used with the SIP account hosted on Odorik.cz. SPA112 VoIP gateway on the other side was receiving the call. An uncompressed PCMA-PCMA codec was used for the voice transfer. Two calls were made, one of 31 minutes and the other of 38 minutes, where the sound traffic dropped out in the last minute or two.

A small amount of usual DNS noise is present in the beginning of the capture, but not during the call.

Microsoft Skype and Teams were a bit more difficult. The first attempt was between the Teams client on my mobile phone and the Skype client on the Fedora client VM. The second call was made between two of my personal Skype accounts.

The VM needed to have Intel HD Audio sound card emulation set up, instead of the default AC97. Then a USB webcam was connected to the VM. The audio usually broke down after several cycles of VM sleep and resume; nevertheless, I successfully captured usable traffic.

[3] http://download.stefan.ubbink.org/ToS/tears_of_steel_1080p.webm.torrent
[4] https://academictorrents.com/details/5d6d0df7ed81efd49ca99ea4737e0ae5e3a5f2e5
[5] https://torrent-file-editor.github.io/

Figure 2.3 Voice capture scenario with SIP capture

As a side note, it was interesting to discover that personal Skype and school Teams could make calls between each other, but the personal Teams account and the school Teams account could not.

Again, some DNS and HTTPS noise was present in the capture, which could not be avoided without loss of functionality.

2.3.3 Exporting the flows

After I captured the data into PCAP files, I then proceeded to export the flows using ipfixprobe. A command exporting the flows looked like this in general.

```
ipfixprobe -i "pcap;file=capture.pcap" \
    -p pstats -p phists -p bstats \
    -s "cache;a=150" \
    -o "unirec;i=f:capture.trapcap:w;p=(pstats,phists,bstats)"
```

Instead of traditional exporting of flows after 300 seconds, I needed to adjust this to 150 seconds, which is what the `cache;a=150` storage plugin does. This was done because I did not have enough flows to perform effective machine learning on longer flows.

Then I used NEMEA's `logger` to export flows into the CSV format, which I could then load with Python and pandas directly.

```
/usr/bin/nemea/logger -i "f:capture.trapcap" \
    --title --write "capture.csv"
```

2.4 WireGuard protocol detection

The result of my work will be a functional prototype of a WireGuard traffic detector. To do that, I need to figure out which properties of traffic can generally be utilized for proper detection. I am building upon the Section 1.2 where I describe the WireGuard protocol in detail.

2.4.1 Transport layer assumptions

WireGuard can be run on any UDP port. As I mentioned in Section 1.2.1, the port may be selected randomly. Therefore, I have to analyze the entirety of UDP traffic.

I also assume that some WireGuard operators might want run their servers on ports 80 (HTTP), 443 (HTTPS), 53 (DNS) and others, to allow connection from networks with strict firewall rules. This is the main reason why it is not possible to easily exclude a range of ports from detection. It is also possible to encounter some false detections on these ports, though I expect that to be rather rare.

2.4.2 Suitable features

For the DPI, it seems that the most promising is the combination of message type and the reserved bytes at the beginning, combined with message lengths, which are fixed for the handshake messages and mostly divisible by 16 in the data messages due to the padding. The shortest possible message length of WireGuard protocol is 32 bytes, which is encrypted data (ID 0x04) with zero bytes of encrypted payload. This is also known as a keepalive packet. If the UDP payload length is smaller than that, the message cannot be a WireGuard message.

If these alone are not enough, then a mechanism of tracking the state machine could be devised.

For the machine learning, IP flow time characteristics will be evaluated as exported by ipfixprobe's PHISTS and PSTATS plugins, and the packet lengths will also be exported. I hope that packet lengths could be used, as well as some of the periodical events described in Section 1.2.5 will influence the histograms and time statistics.

2.5 Developing a naive WireGuard detector for ipfixprobe with DPI

2.5.1 Implementation

I based my implementation on Wu's work around the dissector for Wireshark, combined with my own research of the WireGuard source code. Deep Packet Inspection is used for the analysis of the packets contents.

My plugin attempts to parse every UDP packet from each UDP flow. To shorten the processing time for non-WireGuard packets, the principle of short-circuiting is used, going from the most general to the most specific attributes of the protocol.

First, the UDP payload length is checked. It has to be at least 32 bytes.

Then, the three reserved bytes (2–4) are checked to be zero. The Linux kernel driver of WireGuard also expects these to be zero and would not process the packet otherwise, as it reads and compares the whole 32-bit value.[6]

Then, the packet type is checked, as well as its expected length. I further try to distinguish the flow direction, which is marked by packet's `source_pkt` attribute.

If we get a "session initiation" packet (ID 0x01), two things will happen. First, the random index of the sender is extracted from the packet. Second, that index is compared with the previous index stored in the loaded flow. If it differs, a new flow is created using a `FLOW_FLUSH_WITH_REINSERT` flag. This usually happens after either 120 seconds for the session initiator or $2^{64} - 2^{16} - 1$ messages for the responder (VI. Timers & Stateless UX, A. Preliminaries, [12]).

[6] See the source code at https://github.com/WireGuard/wireguard-linux/blob/42249dba6b4695c53b12545eda4f06eb90dc5ff8/drivers/net/wireguard/receive.c#L553

Table 2.2 The UniRec fields of my WG (WireGuard) ipfixprobe exporter

Output field	Type	Description
WG_CONF_LEVEL	uint8	level of confidence that the flow record is a WireGuard tunnel
WG_SRC_PEER	uint32	random source peer identifier
WG_DST_PEER	uint32	random destination peer identifier

The "cookie" message (ID 0x03) is also recognized, although it was more difficult to gather data to test this properly, and therefore it currently only recognizes the sender index from said message.

2.5.2 UniRec fields

My `wg` plugin exports three UniRec fields (see format description in Section 1.5.1), which are described in Table 2.2. The protocol messages themselves (described in Section 1.2.3) are rather sparse and I have only chosen to export fields which, in my view, contain useful information. The most important is the confidence level (`WG_CONF_LEVEL`), which has a range of 0–100, where:

- 100 means that the flow is very likely to be WireGuard;
- 1 means that the flow shares common characteristics of a WireGuard flow yet it is very likely not WireGuard (false positive). See Section 2.5.3.
- 0 means that the flow is definitely not WireGuard.

2.5.3 False positive detection during operation

During data collection, it was discovered that some DNS packets were erroneously classified as WireGuard traffic. This happens due to their header format and due to the fact that, occasionally, DNS packets may be of the same payload length as WireGuard packets. This would only happen with DNS *queries*, not responses. During the data analysis, I have discovered the *ESET Web Access Protection* queries being wrongly detected as WireGuard traffic, due to them being accidentally in the appropriate format for WireGuard.

To remedy this misclassification, I have modified the parsing algorithm to detect the most common case of DNS queries based on the counts in the record — one (1) question, zero answers. These two are unsigned 16-bit integers in big-endian. The cases with the biggest probability of misdetection are Transaction ID of 0x0300 or 0x0400.

2.5.4 Source code

The source code of my implementation of the detector is available in the attachment of the thesis. It is also present in the master branch of the ipfixprobe repository [7], with a fix waiting to be merged[8].

2.6 Developing a machine learning prototype

Whereas the DPI detector worked with full packet data, the machine learning detector works with IP flow characteristics. It mostly ignores parameters such as IP address, transport protocol

[7] https://github.com/CESNET/ipfixprobe/blob/master/process/wg.cpp
[8] https://github.com/CESNET/ipfixprobe/pull/200

or port. Instead, it focuses on time-related statistics as features. This allows it not to require any packet data, but that also means that I need to discover patterns in the data that may not be readily available. This is why the machine learning is needed here: to train the model that will then be used to classify traffic.

I train my decision-tree-based models with the AdaBoost classifier [41] and with LightGBM [42], a gradient boosting framework. I chose them as they are both widely used and mature, and use a different principle of building the decision tree.

However, the data in the dataset, which is described in the Section 2.3, determine, to a great extent, the features to which the models will be trained.

To allow quick filtering and further work, I populated the flows into pandas' DataFrames, which is a two-dimensional structure holding tabular data. This was needed so I could explore the features effectively.

Then, to see the features and analyze them, I fitted these DataFrames into the Feature Exploration Toolkit (FET) Explorer class instance. The toolkit was helpful in several ways. First, during the fitting, low variance features were removed. These would have no visible effect on training, and they would only make the analysis more difficult. Then, it would allow me to see the correlations between the features. It performed the principal component analysis (PCA) of the data, which allowed to reduce the dimensionality of the data and to discover relations between them. The toolkit was also able to calculate the most important features in the data.

Then, I could plot the best features and how they interacted with each other. It still warranted a set of human eyes to remove features that I deemed too similar (there was a perceived correlation between them in the data). A correlation matrix helped to see where the correlations have been between the features, and if they were related, such as *lengths_max* and *fwd_lengths_max*.

As for the training itself, I divided the data into training set and a validation set. I used a stratified split, to ensure the classes had the same representation in each split, in a 2:1 ratio, with random state 69, by using the `train_test_split` function of the scikit-learn library.

Hyperparameter optimizations were then performed on the training sets. I performed a 3-fold cross-validation and took the mean score of the three attempts.

2.6.1 Hyperparameter search

Optuna studies helped me optimize the hyperparameters. Two different strategies were used: naive for AdaBoost, and a step-wise algorithm for LightGBM. Both are described in [43].

For AdaBoost, I used a naive optimizing function which randomly suggested values from defined ranges for these hyperparameters: `criterion`, `max_depth`, `min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf` and `n_estimators`. This tries to find the best combination, but the total number of possible combinations is a product of the six hyperparameters, thus the search space is quite big.

For LightGBM, I used Optuna's specific hyperparameter tuner for LightGBM with 3-fold cross-validation, `LightGBMTunerCV`. *"It optimizes the following hyperparameters in a stepwise manner: `lambda_l1`, `lambda_l2`, `num_leaves`, `feature_fraction`, `bagging_fraction`, `bagging_freq` and `min_child_samples`."* [44]

For AdaBoost, I was optimizing with the objective of maximizing the F_1 score. For LightGBM, I was instead minimizing the metric of *log loss*.

2.6.2 WireGuard detection

For the binary WireGuard detection, I exported all my captured data into flows. The only label here was "WireGuard or not", thus I combined all the flows together.

The ratio of WireGuard:non-WireGuard flows was generally very skewed (approx. 1:2004). It was necessary to remove some non-WireGuard flows during preprocessing. I excluded flows which described traffic with 3 packets or less. They were too small to reliably detect anything

meaningful, and they were mostly DNS queries and connection attempts. By removing those, and any flows which did not generate any per-packet direction statistics, the ratio dropped to approx. 1:67.

Even though WireGuard runs over UDP, I decided to also include TCP flows in the training set; however, I discarded the TCP-specific features like `{psh,ack,fin}_{ratio,count}` which describe TCP flags.

Feature exploration revealed some correlated features. During the first experiments with only UDP traffic, the features that correlated the most had a coefficient of 0.96, 0.97 etc. But after I added the TCP flows into the mix, most correlations fell under 0.95. I have thus removed one of the feature pair that had the ratios somewhere around 0.94, 0.93, 0.92. If `pkt_iat_std` and `pkt_iat_max` were correlating, then I usually removed the `max`. The notebook `wg-nonwg-1-explore.ipynb` contains the full list.

For AdaBoost, I performed several experiments with various hyperparameter optimizations. Eventually the studies of hyperparameters with the 3-fold cross-validation resulted in the training set started getting F_1 score of 1.0, which was weird, as it could be a sign of overfitting and would not be usable with other traffic. The validation set also got a F_1 score of 1.0. However the study took several hours to finish.

For LightGBM, I experimented with the value of `max_depth` parameter. Raising it from default 7 to 10 resulted in an increase of precision, after optimizing the hyperparameters.

2.6.3 Traffic category detection

The detection of traffic categories required the use of multi-class classification algorithms. I performed it only on the WireGuard data.

Both AdaBoost and LightGBM performed rather comparably, and the hyperparameter optimization increased the precision and recall significantly. At the point of F_1 score around 0.9, however, the gains were no longer that great.

For LightGBM, I once again experimented with the value of `max_depth` parameter. Raising it brought some improvement of F_1 score on the training set, but not in the validation set.

As a part of an experiment, I also increased `max_leaves` from 7 to 10, which, after optimizing, resulted in an increase of precision.

2.6.4 Result models

In the end, I performed the fitting and validation with the following hyperparameters, which were found during the hyperparameter search using Optuna optimizing functions.

See the code listings for the parameters that ultimately performed the best — Code listing 2.14, Code listing 2.15, Code listing 2.16 and Code listing 2.17.

First, I will reveal the final scores of the WireGuard detection.

Code listing 2.14 AdaBoost hyperparameters for WireGuard detection

```
{
    'criterion': 'gini',
    'max_depth': 31,
    'min_samples_leaf': 162,
    'min_samples_split': 149,
    'min_weight_fraction_leaf': 0.13931986457748236,
    'n_estimators': 225
}
```

Code listing 2.15 LGBM hyperparameters for WireGuard detection

```
{
    'objective': 'binary',
    'metric': 'binary_logloss',
    'deterministic': True,
    'verbosity': -1,
    'max_depth': 10,
    'feature_pre_filter': False,
    'lambda_l1': 0.0,
    'lambda_l2': 0.0,
    'num_leaves': 31,
    'feature_fraction': 0.5,
    'bagging_fraction': 0.620877224888789,
    'bagging_freq': 2,
    'min_child_samples': 20
}
```

Code listing 2.16 AdaBoost hyperparameters for traffic class detection

```
    {
        'criterion': 'log_loss',
        'max_depth': 191,
        'min_samples_split': 2,
        'min_samples_leaf': 2,
        'min_weight_fraction_leaf': 3.751430785131106e-05,
        'n_estimators': 222
    }
```

AdaBoost model with the best F_1 macro average was 1.0, precision 1.0 and recall 1.0 after validating on a validation set.

LightGBM, with `max_depth` of 10 did not fall far off, with F_1 score of 0.998168, precision 0.996403 and recall 0.999945.

I took a quick look at the wrongly matched flows, with indexes 1,678,923 (web) and 1,189,380 (email). Both were predicted as WireGuard but were ultimately not. Both had quite small amount of packets (web had 4 and email had 15 and their packets were on the larger side. I cannot conclusively however say where the model mispredicted.

Between AdaBoost and LighGBM in these instances, the difference was minimal in all the attempts.

And now for the traffic class detection. My best AdaBoost model has got a resulting F_1 score of 0.918831, with precision 0.923785 and recall 0.917808. However the maximum depth of the tree is really high.

LightGBM model with the best parameters had the F_1 score of 0.909869, with precision score 0.915427 and recall score: 0.907524, and the tree had maximum depth set to default.

The code used to was too long to include in the main part and is available in the attachment of the thesis, as well as most of the studies performed.

Code listing 2.17 LGBM hyperparameters for traffic class detection

```
{
    'objective': 'multiclass',
    'metric': 'multi_logloss',
    'deterministic': True,
    'n_jobs': 16,
    'verbosity': -1,
    'num_class': 7,
    'feature_pre_filter': False,
    'lambda_l1': 0.07165881099222629,
    'lambda_l2': 0.026475595850891546,
    'num_leaves': 21,
    'feature_fraction': 0.44800000000000006,
    'bagging_fraction': 0.9598010723030449,
    'bagging_freq': 7,
    'min_child_samples': 20
}
```

Chapter 3
Evaluation and testing

First, I will present a summary of the numbers of data captures. With them in mind, I will then determine how accurate my DPI detector actually is. After that, I will test its throughput. Furthermore, I will apply the same to the machine learning models, albeit with different inputs — extracted flows instead of full traffic captures. And at the end of the chapter, I will discuss the results.

3.1 Captured datasets

During the development of the DPI module, I created a small dataset to verify that the detection works as intended. This dataset only contains a small sample that should always be detected as WireGuard, and one DNS flow that is more DNS than WireGuard, but should also be detected by the plugin.

To test the detector properly, I also collected a bigger dataset. A total of 42 captures were performed across all categories, making a grand total of 32.85 GB of inner traffic and 35.67 GB of WireGuard traffic. This was then exported into 1,678,216 non-WireGuard flows and 838 WireGuard flows. The amount of flows exported by ipfixprobe for each category, and the volume of data captured per category, is summarized in Table 3.1.

The dataset that contained only inner flows, is marked in the attachments as Dataset A. The dataset with outer flows as well is marked as Dataset B. Both are supplemental attachments to the thesis.

Table 3.1 Total numbers of flows and data collected per category

Category	WG Flows	WG data	Non-WG flows	Non-WG data
chat	151	137.90 MB	692	123.00 MB
email	115	1,496.04 MB	540	1,382.84 MB
file	186	6,355.38 MB	137	5,971.33 MB
p2p	129	20,199.10 MB	1,649,860	18,514.43 MB
video	191	3,801.25 MB	3,867	3,554.90 MB
voice	143	2,702.43 MB	2,169	2,408.71 MB
web	140	981.31 MB	20,951	897.72 MB
Total	* **1,055**	**35.67 GB**	**1,678,216**	**32.85 GB**

* Note that for WireGuard flows processed with my DPI plugin `wg`, the maximum duration of a WireGuard flow will be 120 seconds, as I describe in Section 2.5.1. With that in mind, the number of exported WireGuard flows becomes 1,055. In all other cases (except for throughput

Code listing 3.1 The result of ipfixprobe's make check

```
[bpuser@fedora ipfixprobe]$ make check
...
make[5]: Entering directory '/home/bpuser/sources/ipfixprobe/...'
...
PASS: wg.sh
...
============================================================================
Testsuite summary for ipfixprobe 4.9.2
============================================================================
# TOTAL: 23
# PASS:  23
...
```

testing), it should be 838, as I split the exported flows after 150 seconds, instead of regular 300 seconds.

3.2 Testing of precision of the detectors

3.2.1 The DPI detector

After collecting data, I have a considerable dataset to run my tests on, labeled by the category of traffic and whether the traffic is in a WireGuard tunnel or not. I shall test that assumption here, and evaluate any inconsistencies that may happen.

I will first convert the datasets to flows with my `wg` plugin enabled, then convert it with Nemea's `logger` to CSV, and verify whether the flows contain the WireGuard fields. If they do, then I will check the confidence level. I will only consider a flow to be WireGuard if the confidence level is exactly 100, per the current implementation.

To ensure that non-WG flows do not get discarded, I also needed to add the PHISTS plugin to the exporter pipeline.

Test 1: Testing dataset for ipfixprobe module

This is a unit test in the ipfixprobe repository. A testing PCAP is included in the `pcaps/wg.pcap`, which exports to 13 flows that should all identify as WireGuard. The reference contents of the export are available in `tests/functional/reference/wg`. One of them is DNS masquerading as WireGuard, the rest are valid WireGuard flows. Code listing 3.1 shows that I am getting the expected result.

Test 2: WG-only dataset

The expected result is that all flows are detected as WireGuard. Because there is no other noise in this data, other than WireGuard flows, there should be a 100% true positive rate.

The results in Table 3.2 confirm that the results were, indeed, a 100% true positive rate, for a total of 1055 WireGuard flows.

Test 3: Non-WG-only datasets

The expected result is that all such flows are detected as *not* WireGuard.

The result of the tests is shown in Table 3.3. All of the 1,678,216 flows were correctly detected as *not* being WireGuard.

Table 3.2 Results of flow matching for outer, WireGuard traffic

Label	Flows	WG	Likely not WG
chat	151	151	0
email	115	115	0
file	186	186	0
p2p	129	129	0
video	191	191	0
voice	143	143	0
web	140	140	0
Total	**1,055**	**1,055**	**0**

Table 3.3 Results of flow matching for inner, non-WireGuard traffic

Label	Flows	WG	Likely not WG
chat	692	0	0
email	540	0	0
file	137	0	0
p2p	1,649,860	0	0
video	3,867	0	0
voice	2,169	0	0
web	20,951	0	0
Total	**1,678,216**	**0**	**0**

3.2.2 The machine learning detector – WG detection

As I explained in Section 2.6, I used the AdaBoost algorithm ensemble and Light GBM framework to train my models. The decision which the model needs to make here is binary: is the traffic WireGuard, or is it not.

First, let's show the final AdaBoost model.

After training the model with the training set, I then validated with the validation set. The resulting confusion matrix in Table 3.4 indicates that I had 100% precision and recall on the validation slice as well.

Table 3.4 Validation set evaluation for AdaBoost model

True \ Predicted	0	1
0	18394	0
1	0	276

Next, I have the Light GBM, with the Optuna optimization library also used here.

Table 3.5 shows the result of the model with optimized hyperparameters.

Table 3.5 Validation set evaluation for LightGBM model

True \ Predicted	0	1
0	18392	2
1	0	276

3.2.3 The machine learning detector – traffic class detection

For this, I have used the same algorithms as in the previous section, but only trained on the WireGuard flows.

First, I have run the predictions on the validation dataset, using my best AdaBoost model. The resulting confusion matrix is displayed in Figure 3.1.

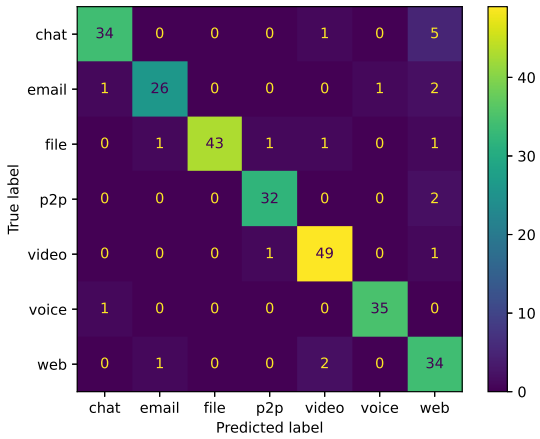

Figure 3.1 Confusion matrix of the AdaBoost model for traffic class detection

For LightGBM, I also validated the corresponding best model with the validation data. The result is in Figure 3.2.

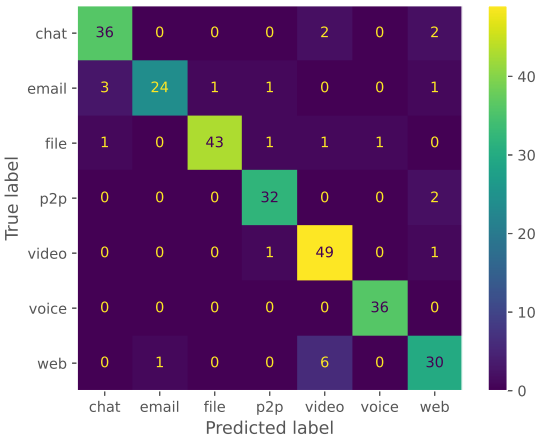

Figure 3.2 Confusion matrix of the LightGBM model for traffic class detection

Code listing 3.2 Command to test throughput of DPI detector `wg`

```
# Merge the traffic
ls -1 cap*/*out_wan.pcap | xargs mergecap -F pcap \
    -w /tmp/throughput_out.pcap
ls -1 cap*/*in_wg.pcap | xargs mergecap -F pcap \
    -w /tmp/throughput_internal.pcap

# Test without wg module
time ipfixprobe -i "pcap;file=/tmp/throughput_out.pcap" \
    -p phists \
    -o "unirec;i=f:/tmp/tmpfile.trapcap:w;p=(phists)"

# Test with wg module
time ipfixprobe -i "pcap;file=/tmp/throughput_out.pcap" \
    -p phists -p wg \
    -o "unirec;i=f:/tmp/tmpfile.trapcap:w;p=(phists,wg)"
```

3.3 Throughput testing of the DPI detector

I performed throughput testing on the DPI version of the detector, which exports the flows. I performed a series of tests of the current ipfixprobe detector compiled with my WireGuard plugin.

I have tested by reading two combined captured samples:

- the combined flows within the tunnel, and
- the combined traffic of the WireGuard-encapsulated traffic.

I tested two times for each: with and without my `wg` plugin in the processing chain. First, I loaded the sample to RAM, to eliminate any I/O delays as much as possible. Then, I ran an ipfixprobe scenario 10 times in a row, testing both flows with `wg` plugin and without, and recorded the time for each ipfixprobe run. The commands used are specified in Code listing 3.2.

Other activities on the computer were suspended, and the power profile of the host was set to Performance. The timing itself was done on the same VM on which I've done my packet captures. The CPU was AMD Ryzen 7 7840U and the VM had 4 vCPU cores and 42 GB of RAM available.

The commands described in Code listing 3.2 were timed with the integrated `time` command in bash shell. I exported the output of the `time` command by the *real*, *sys* and *user* components.

The results can be seen in Table 3.6.

Table 3.6 Time of processing for DPI ipfixprobe plugin

Dataset	Packets	Biflows	Plugins	User time – mean average
WireGuard	36,480,788	432	phists	11.795 s (±0.150)
	36,480,788	1,052	phists, wg	13.000 s (±0.100)
Inner traffic	36,477,030	1,678,339	phists	20.236 s (±0.410)
			phists, wg	20.729 s (±0.192)

3.4 Throughput testing of ML detectors

CPU used for testing was AMD Ryzen 7 7840U, power profile was set to Performance, with limited number of other applications running on the machine.

The predictions of LightGBM were limited to 1 CPU core. LightGBM was able to utilize all 8 cores (16 with hyperthreading) of my machine, which would have made the results hard to compare.

For the WireGuard detection, I ran the prediction on the same dataset which was used for training and validation. For the traffic classes models, it was the same, but only non-WireGuard flows were included. To have measurable differences, I had to multiply the dataset several hundred (or thousand) times, and then clip it to 2 million flows.

See Table 3.7 and Table 3.8 for the results.

Table 3.7 Time of processing for WireGuard models

ML Model	Flows	Mean average
AdaBoost	2,000,000	35.360 s (±0.677)
LightGBM	2,000,000	2.597 s (±0.066)

Table 3.8 Time of processing for traffic classes models

ML Model	Flows	Mean average
AdaBoost	2,000,000	46.631 s (±0.033)
LightGBM	2,000,000	27.616 s (±0.231)

3.5 Discussion

As I demonstrated in Section 3.2.1, the WireGuard detector was able to detect 100% of my captured WireGuard traffic, and, at the same time, it correctly detected that *none* of the traffic in the inner dataset was WireGuard, which was the expected result, due to the simplicity of the protocol, and the recognizable signature of WireGuard packets.

However, it also has some weaknesses. You can pass WireGuard through TCP, though this is not officially supported. To save some processing time, the DPI detector only detects WireGuard over UDP packets. The machine-learned models could theoretically also detect WireGuard over TCP, if the traffic behaves in a similar way to other WireGuard packets.

To my surprise, the machine-learned models, which I trained on my collected dataset, were also able to detect WireGuard quite reliably on the provided validation set, only from the intra-packet time distribution statistics, with limited false positives or false negatives. As WireGuard internally utilizes timers to keep the connection alive and to rotate keys periodically, I find it very likely that this contributes to the positive result. My client setting of `PersistentKeepalive = 25` may thus be helping that, and it would make sense to collect some data without it, which I did not think of at the time.

On the other hand, I collected very few examples where one peer cannot reach the other peer, and there is none in the collected dataset. The machine-learned model will, thus, very likely miss these flows.

If I talk about the models for classes, even from such a small sample, the mispredictions attributing the web class to some other classes, or vice versa, were not very surprising, as the web browsing contained a lot of different kind of content, including chat. Chat in the Matrix protocol world happens over HTTPS protocol, so it did not surprise me too much that there were mispredictions there, and quite repeated.

Throughput-wise, my WireGuard DPI detector `wg` added an overhead of around 10% in the case where all the processed traffic was WireGuard, and around 1% when there was no WireGuard traffic, although the second value is within the standard deviation of my measurement. Machine-learned models, however, defied expectations. For the same task, the performance of the two

models was different under similar conditions. The difference is extremely visible on the side of WireGuard detection models. The model differences could be explained by varying decision tree depths and differences in algorithms, but as I can see in Table 3.7, the difference in this binary case is striking.

However, for all of the machine-learned detectors, the collected dataset is very small and needs further collection of data to properly validate the model, with multiple clients involved, possibly other operating systems, and the TCP WireGuard also involved. If I also wanted some real-life non-WireGuard data, I could have used the ISCXVPN2016 [4] dataset, which could have given me a great corpus for the non-VPN data. However, I ran out of time.

When I think beyond the bugs in implementation, I can see a possibility of false negatives due to manipulations with data (such as Man-in-the-Middle attacks, or network errors). Currently, my WireGuard DPI detector will stop analyzing any flow where the type is outside of the known types, the message types don't match their known lengths, or the data message length is smaller than the minimum. If I compare my code to the real kernel driver, then the length and type checks are similar. However, my code will also mark any flow that fails at least one of these conditions at least once as *definitely not WireGuard*, which, I think, is probably not the best approach.

Chapter 4
Conclusion

This bachelor's thesis studied the WireGuard protocol and how I could detect it in network traffic. This is important for a number of reasons, network defense and protection notwithstanding, where network administrators want to have the ability to detect VPNs.

I studied the protocol and the prior work, and collected some example data. Then, I was able to implement a detection algorithm utilizing deep packet inspection, utilizing mainly the well-defined structure and lengths of the packet.

Furthermore, I designed and implemented a plugin for ipfixprobe utilizing said deep packet inspection.

Next, I tried to gather much larger dataset, consisting of multiple categories of traffic, both WireGuard and unencrypted. This gave me a corpus which I could then use to validate the detector, and to also attempt methods of machine learning.

This gave rise to four additional detection prototypes: two, which tested for presence of WireGuard, and another two, which classified encrypted IP flow data into categories. These used detection based on IP flow characteristics. They were implemented as machine-learned models, and between each two, different algorithms were used for training the models.

At the end, I tested my ipfixprobe plugin and the machine-learned models for precision and throughput. During precision testing, I was interested mainly in the number of false positives for the WireGuard detection, and general misclassifications for the traffic classes. The throughput was then tested in a way to estimate the impact of the plugin on flow exporting with and without my plugin. I also tested the throughput of machine-learned models.

In the short discussion of the results, I hinted at areas of possible improvement of the detectors. Further work could improve the reliability of the detector in situations where the data is not clean, and otherwise expand the range of detectable traffic.

Bibliography

1. DOROSHENKO, Dmitriy. *New Protocols for Virtual Private Networks* [online]. 2020. [visited on 2020-12-04]. Available from: http://hdl.handle.net/10467/88058. Master's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering.
2. MACKEY, Steven; MIHOV, Ivan; NOSENKO, Alex; VEGA, Francisco; CHENG, Yuan. A Performance Comparison of WireGuard and OpenVPN. In: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 162–164. CODASPY '20. ISBN 9781450371070. Available from DOI: 10.1145/3374664.3379532.
3. ČTRNÁCTÝ, Martin. *Softwarový modul pro rozpoznání VPN v síťovém provozu* [online]. 2020. [visited on 2020-12-06]. Available from: http://hdl.handle.net/10467/87996. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology.
4. DRAPER-GIL, Gerard; LASHKARI, Arash Habibi; MAMUN, Mohammad Saiful Islam; GHORBANI, Ali A. Characterization of Encrypted and VPN Traffic Using Time-Related Features. In: *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*. 2016, pp. 407–414.
5. NAAS, Mohamed; FESL, Jan. A novel dataset for encrypted virtual private network traffic analysis. *Data in Brief*. 2023, vol. 47, p. 108945. ISSN 2352-3409. Available from DOI: https://doi.org/10.1016/j.dib.2023.108945.
6. SANDQUIST, Christoffer; ERSSON, Jon-Erik. *Towards Realistic Datasets forClassification of VPN Traffic : The Effects of Background Noise on Website Fingerprinting Attacks*. 2023.
7. MASON, Andrew G.; STIFFLER, Rick. *Cisco Secure Virtual Private Networks*. Cisco Press, 2001. ISBN 1587050331.
8. JAHA, Ahmed A.; SHATWAN, Fathi Ben; ASHIBANI, Majdi. Proper Virtual Private Network (VPN) Solution. In: *2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*. 2008, pp. 309–314. Available from DOI: 10.1109/NGMAST.2008.18.
9. KHANVILKAR, S.; KHOKHAR, A. Virtual private networks: an overview with performance evaluation. *IEEE Communications Magazine*. 2004, vol. 42, no. 10, pp. 146–154. Available from DOI: 10.1109/MCOM.2004.1341273.
10. KENT, Stephen. *IP Encapsulating Security Payload (ESP)* [RFC 4303]. RFC Editor, 2005. Request for Comments, no. 4303. Available from DOI: 10.17487/RFC4303.
11. PRINCE, Matthew. *WARP is here (sorry it took so long)* [online]. Cloudflare, 2019-09-25. [visited on 2020-12-06]. Available from: https://blog.cloudflare.com/announcing-warp-plus/.

12. DONENFELD, Jason A. WireGuard: Next Generation Kernel Network Tunnel. In: *NDSS*. 2017.
13. CLOUDFLARE. *BoringTun, a userspace WireGuard implementation in Rust* [online]. Cloudflare, 2019-03-27. [visited on 2024-01-06]. Available from: https://blog.cloudflare.com/boringtun-userspace-wireguard-rust.
14. DOWLING, Benjamin; PATERSON, Kenneth G. A cryptographic analysis of the WireGuard protocol. In: *International Conference on Applied Cryptography and Network Security*. Springer, 2018, pp. 3–21.
15. DONENFELD, Jason A; MILNER, Kevin. *Formal verification of the WireGuard protocol*. 2017-07. Tech. rep.
16. LIPP, Benjamin; BLANCHET, Bruno; BHARGAVAN, Karthikeyan. A mechanised cryptographic proof of the WireGuard virtual private network protocol. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 231–246.
17. DONENFELD, Jason A. *Known Limitations - WireGuard* [online]. ZX2C4, Edge Security. [visited on 2024-01-06]. Available from: https://www.wireguard.com/known-limitations/.
18. DONENFELD, Jason A. *WireGuard: a new VPN tunnel (discussion)* [online]. LWN.net, 2016-08-23. [visited on 2021-12-18]. Available from: https://lwn.net/Articles/697943/.
19. DONENFELD, Jason A. *Quick Start - WireGuard* [online]. ZX2C4, Edge Security. [visited on 2021-12-18]. Available from: https://www.wireguard.com/quickstart/.
20. *Tunneling WireGuard over TCP with TunSafe* [online]. TunSafe AB. [visited on 2024-01-06]. Available from: https://tunsafe.com/user-guide/tcp.
21. BERNSTEIN, D. J. *TAI64, TAI64N, and TAI64NA* [online]. 1997-07. [visited on 2024-01-06]. Available from: https://cr.yp.to/libtai/tai64.html.
22. WU, Peter. *Analysis of the WireGuard protocol* [online]. 2019. [visited on 2020-12-06]. Available from: https://lekensteyn.nl/files/pwu-wireguard-thesis-final.pdf. Master's thesis. Eindhoven University of Technology, Eindhoven, Netherlands.
23. WU, Peter. *Wireshark Bug Database – Support for WireGuard protocol* [online]. 2018-07-26. [visited on 2022-01-24]. Available from: https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=15011.
24. GMBH, ipoque. *Rohde & Schwarz Adds Emerging WireGuard VPN Protocol to its Deep Packet Inspection (DPI) Software Library, R&S®PACE 2* [online]. 2019-01-23. [visited on 2022-01-24]. Available from: https://www.ipoque.com/news-media/press-releases/rohde-amp-schwarz-adds-emerging-wire-guard-vpn-protocol-to-its-deep-packet-inspection-dpi-software-library-r-amp-s-pace-2.
25. DHARMAPURIKAR, S.; KRISHNAMURTHY, P.; SPROULL, T.; LOCKWOOD, J. Deep packet inspection using parallel Bloom filters. In: *11th Symposium on High Performance Interconnects, 2003. Proceedings.* 2003, pp. 44–51. Available from DOI: 10.1109/CONECT.2003.1231477.
26. AITKEN, Paul; CLAISE, Benoît; TRAMMELL, Brian. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information* [RFC 7011]. RFC Editor, 2013. Request for Comments, no. 7011. Available from DOI: 10.17487/RFC7011.
27. HOFSTEDE, R.; ČELEDA, P.; TRAMMELL, B.; DRAGO, I.; SADRE, R.; SPEROTTO, A.; PRAS, A. Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *IEEE Communications Surveys & Tutorials*. 2014, vol. 16, no. 4, pp. 2037–2064. Available from DOI: 10.1109/COMST.2014.2321898.
28. EDDY, Wesley. *Transmission Control Protocol (TCP)* [RFC 9293]. RFC Editor, 2022. Request for Comments, no. 9293. Available from DOI: 10.17487/RFC9293.

29. POSTEL, J. *User Datagram Protocol* [RFC 768]. RFC Editor, 1980. Request for Comments, no. 768. Available from DOI: 10.17487/RFC0768.
30. CLAISE, Benoît. *Cisco Systems NetFlow Services Export Version 9* [RFC 3954]. RFC Editor, 2004. Request for Comments, no. 3954. Available from DOI: 10.17487/RFC3954.
31. HOLDREGE, Matt; SRISURESH, Pyda. *IP Network Address Translator (NAT) Terminology and Considerations* [RFC 2663]. RFC Editor, 1999. Request for Comments, no. 2663. Available from DOI: 10.17487/RFC2663.
32. CESNET, z. s. p. o. *NEMEA: System for network traffic analysis and anomaly detection* [online]. CESNET, z. s. p. o., 2020-12-06. [visited on 2020-12-06]. Available from: https://nemea.liberouter.org.
33. ČEJKA, Tomáš; BARTOŠ, Václav; ŠVEPEŠ, Marek; ROSA, Zdeněk; KUBÁTOVÁ, Hana. NEMEA: A framework for network traffic analysis. In: *2016 12th International Conference on Network and Service Management (CNSM)*. 2016. Available from DOI: 10.1109/CNSM.2016.7818417.
34. CESNET, z. s. p. o. *GitHub – CESNET/ipfixprobe README* [online]. 2022-12-20. [visited on 2023-03-23]. Available from: https://github.com/CESNET/ipfixprobe/blob/32f0081d2e20ec4b7fe7470b546fd7dbd7a1295a/README.md.
35. SEGARAN, T. *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. O'Reilly Media, 2007.
36. MOHRI, Mehryar; ROSTAMIZADEH, Afshin; TALWALKAR, Ameet. *Foundations of Machine Learning*. 2nd. The MIT Press, 2018. ISBN 0262039400.
37. LOH, Wei-Yin. Classification and regression trees. *WIREs Data Mining and Knowledge Discovery*. 2011, vol. 1, no. 1, pp. 14–23. Available from DOI: https://doi.org/10.1002/widm.8.
38. FERRI, C.; HERNÁNDEZ-ORALLO, J.; MODROIU, R. An experimental comparison of performance measures for classification. *Pattern Recognition Letters*. 2009, vol. 30, no. 1, pp. 27–38. ISSN 0167-8655. Available from DOI: https://doi.org/10.1016/j.patrec.2008.08.010.
39. DONENFELD, Jason A. *Manual page of wg-quick (8)* [online]. ZX2C4, Edge Security, 2020-07-28. [visited on 2022-01-19]. Available from: https://git.zx2c4.com/wireguard-tools/about/src/man/wg-quick.8.
40. *Segmentation Offloads – The Linux Kernel documentation* [online]. The kernel development community. Version 5.16.0 [visited on 2022-01-19]. Available from: https://www.kernel.org/doc/html/v5.16/networking/segmentation-offloads.html.
41. FREUND, Yoav; SCHAPIRE, Robert E. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*. 1997, vol. 55, no. 1, pp. 119–139. ISSN 0022-0000. Available from DOI: https://doi.org/10.1006/jcss.1997.1504.
42. KE, Guolin; MENG, Qi; FINLEY, Thomas; WANG, Taifeng; CHEN, Wei; MA, Weidong; YE, Qiwei; LIU, Tie-Yan. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In: GUYON, I.; LUXBURG, U. Von; BENGIO, S.; WALLACH, H.; FERGUS, R.; VISHWANATHAN, S.; GARNETT, R. (eds.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, vol. 30, pp. 3149–3157. Available also from: https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.
43. OZAKI, Kohei. *LightGBM Tuner: New Optuna Integration for Hyperparameter Optimization* [online]. Optuna, 2020-03-03. [visited on 2024-01-06]. Available from: https://medium.com/optuna/lightgbm-tuner-new-optuna-integration-for-hyperparameter-optimization-8b7095e99258.

44. *optuna.integration. lightgbm.LightGBMTuner – Optuna 3.5.0 documentation* [online]. 2023. [visited on 2024-01-05]. Available from: https://optuna.readthedocs.io/en/v3.5.0/reference/generated/optuna.integration.lightgbm.LightGBMTuner.html.

Contents of the attachment

- `readme.txt` ..... brief contents of the medium
- `src`
  - `analysis` ..... Python environment for data analysis
  - `detector-dpi` ..... the implementation of DPI detector
  - `vm-server` ..... scripts for server virtual machine provisioning
  - `vm-client` ..... scripts for client virtual machine provisioning
  - `thesis` ..... source code of the thesis in LaTeX format

The datasets are uploaded at Zenodo under these identifiers:

- Dataset A: 10.5281/zenodo.10492150
- Dataset B: 10.5281/zenodo.10491462