FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Analysis of Tor client behavior and its identification
Student:	Bc. Tibor Engler
Supervisor:	Ing. Tomáš Čejka, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2021/2022

Instructions

Investigate the Tor network protocol and its traffic, and also study the flow-based network traffic monitoring and analysis.
Survey existing scientific works regarding Tor traffic analysis and client deanonymization.
Design and create a testing laboratory environment to study traffic correlation between Tor endpoints.
Design an algorithm for Tor traffic analysis based on the observation of communication in the laboratory environment.
Evaluate possibilities of the analysis and estimation of sources and targets of Tor communication.

—

Wilfried Mayer, Georg Merzdovnik and Edgar Weippl: Actively Probing Routes for Tor AS-level Adversaries with RIPE Atlas. SEC2020.
Asya Mitseva, Marharyta Aleksandrova, Thomas Engel and Andriy Panchenko: Security and Performance Implications of BGP Rerouting-resistant Guard Selection Algorithms for Tor
Hayes, Jamie, and George Danezis. 'k-fingerprinting: A robust scalable website fingerprinting technique.' 25th USENIX Security Symposium (USENIX Security 16). 2016.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Analysis of Tor Client Behavior and Its Identification

Bc. Tibor Engler

Department of Information Security
Supervisor: Ing. Tomáš Čejka, Ph.D.

May 5, 2021

Acknowledgements

Declaration

In Prague on May 5, 2021   . . . .

Citation of this thesis

Abstrakt

Projekt Tor je všeobecne známa anonymizačná technológia. Komunikácia pomocou Tor-u je založená na niekoľkých vrstvách šifrovania a náhodnom smerovaní, ktoré by mali garantovať vysokú úroveň súkromia. Táto práca však ukazuje, že súkromie Tor-u je značne obmedzené, pokiaľ sledovacie systémy začnú využívať techniky hlbokého učenia a sledujú pri tom komunikáciu klienta aj jeho cieľu. Výstupom tejto práce je klasifikačný model na báze hlbokého učenia (konvolučná neurónová sieť), ktorý bol vyhodnotený pomocou komplexných experimentov na nových dátových sadách, zachytených na reálnej sieti. Navrhovaný algoritmus dokáže rozhodnúť (s presnosťou vyššou ako 90 %), či dve spojenia pozorované na rôznych miestach predstavujú jednu a tú istú komunikáciu medzi klientom a serverom. Pozitívny výsledok môže odhaliť konkrétny cieľ, s ktorým klient komunikuje, a teda predstavuje možnú bezpečnostnú slabinu, oslabujúcu celý proces anonymizácie. V porovnaní s existujúcimi prácami je nami vyvinutý model schopný pracovať s rozšírenými údajmi o IP toku namiesto úplných paketových záznamov.

Kľúčové slová   Tor, onion routing, hlboké učenie, konvolučné neurónové siete, monitoring sieťových tokov, korelačný útok

Abstract

The Tor project is a well-known anonymization technology. The communication using Tor is based on multiple layers of encryption and randomized routes, which should guarantee a high level of privacy. However, this thesis shows that the privacy of Tor is quite limited when surveillance systems adopt deep learning techniques and observe both client's and target's traffic. The outcome of this thesis is a Deep Learning based classification model (Convolutional Neural Network) that was evaluated using comprehensive experiments with new Tor datasets captured from real network traffic. The proposed algorithm is able to decide (with accuracy higher than 90%), whether the two connections observed in different places belong to the same communication between the client and the server. The positive result discloses a particular target of a client, thus represents a possible privacy weakness of the whole anonymization process. Compared to the existing works, the developed model is able to work with extended IP flow data, instead of full packet traces.

Keywords   Tor, onion routing, deep learning, convolutional neural networks, network flow monitoring, correlation attack

Contents

List of Figures

List of Tables

Introduction

We live in an era in which our privacy is slowly ceasing to exist. When we work, learn, or have fun on the Internet, each of our steps is quietly followed by trackers of advertising companies, which offer us personalized advertising based on our behavior. Among other things, our actions are being monitored by individual states in order to detect potential crimes on the Internet and, in some countries, by oppression regimes that use censorship to control which sites are accessible to their citizens.

Recently, however, there has been increasing talk about protecting our privacy, and so this awareness is spreading in lay society as well. Users who had no idea about encrypting the Internet connection a few years ago now use a VPN to secure their browsing. Furthermore, an increasing number of users are reaching for the use of the most popular anonymization network – Tor. Thanks to its simple user interface without the need to set anything, it is a suitable tool for hiding communication from the eyes of everyone else. With the advent of machine learning, however, the previous statement is becoming less and less true.

The motivation of this work is to challenge the privacy strengths provided by the Tor technology and to evaluate the possibilities of modern machine learning approaches. We will introduce the reader to the principles on which Tor works and show him that by the power of a machine-learned correlation function, it is possible with a high probability to detect anonymized communication sent through Tor.

In the first three chapters, we will focus on the technologies and ideas on which our experiment is based: Tor, Convolutional Neural Networks, and Network flow monitoring. Chapter four is concerned with the design of a laboratory in which we perform a series of experiments. We will describe the processes and decisions we made during the data collection and evaluation process, so the reader will understand what reflections brought us to a particular decision. Next, we will deal with the setting of individual parameters to achieve the highest possible accuracy of our Deep Learning algorithm.

Finally, according to the results of our experiments, which will be comprehensively described in the last chapter, there is a significant chance for adversary entities to identify whether a Tor client is connected to a particular target. Naturally, our experiments were designed to cover a rather special case where some monitoring/surveillance mechanisms can observe the traffic on both client's and server's side. However, this case was studied using real network traffic and environment. As a result, there is a potential privacy risk caused by so-called side channels, i.e., the behavioral characteristics of the network connections given by the applications and their users.

Tor

Tor is an open-source low-latency communication service that allows its users to communicate and browse the Internet anonymously. The name Tor originated as an acronym for "The Onion Router", which was initially sponsored by the United States Navy. However, now it is a non-profit project built on the basis of volunteer servers, the main purpose of which is to create a secure and free environment with built-in functions for the protection of personal data.

Tor has a quite wide range of uses. Most users use it to prevent their ISPs and commercial servers from tracking their browsing preferences, but there are certain groups of people whose lives depend on whether Tor properly anonymizes their browsing. Users in regimes that censor certain websites can use Tor to bypass the censorship. Whistleblowers, activists, and investigative journalists that are in danger of being exposed can use Tor to safely browse the Internet and exchange sensitive information without arousing suspicion. Tor is, however, also used by people on the other side of the spectrum: criminals, hackers, dealers, and other people violating the law use Tor for the same reason – to successfully hide from investigators and the police.

1.1 History

Tor started as a research project with the goal to use the Internet with as much privacy as possible. The initial idea was to route traffic through multiple servers and encrypt it each step along the way so that no one monitoring the network would be able to reveal who is talking to whom.

From its inception in the 1990s, onion routing was conceived to rely on a decentralized network. The network needed to be operated by entities with diverse interests and trust assumptions, and the software needed to be free and open to maximize transparency and separation. That is why in October 2002, when the Tor network was initially deployed, its code was released under

a free and open software license. By the end of 2003, the network had about a dozen volunteer nodes, mostly in the U.S., plus one in Germany [1].

In 2004, the development continued under funding from the Electronic Frontier Foundation. In this year, the location hidden services were deployed[1] and a paper *Tor: The Second-Generation Onion Router* [2] was presented on the 13th USENIX Security Symposium, which laid the ground for the design of Tor we use still today. By the end of the year, there were over 100 Tor nodes on three continents, contributing to the network.

In 2006, the Tor Project, Inc., a nonprofit organization, was founded to maintain Tor's development, which attracted sponsors such as the U.S. International Broadcasting Bureau, Internews, Human Rights Watch, the University of Cambridge, Google, and Netherlands-based Stichting NLnet to fund the project [3].

As Tor gained popularity, its users started demanding that its creators address censorship by allowing those living under oppressive governments to publish their thoughts and access restricted websites freely. This motivated Tor's creators to start developing a way for its network to get around government firewalls in 2007 so its users could access government-restricted websites [4].

Although the authors' motives were noble, only technical fans could join the Tor network at the time, as the setup was complicated for the standard users. This situation led the authors to develop a more user-friendly version of Tor – the Tor Browser, whose development began in 2008.

As stated in [1], with Tor Browser having made Tor more accessible to everyday Internet users and activists, Tor was an instrumental tool during the Arab Spring beginning in late 2010. It not only protected people's identity online but also allowed them to access critical resources, social media, and websites that were blocked.

The need for tools safeguarding against mass surveillance became a mainstream concern thanks to the Snowden revelations in 2013. Not only was Tor instrumental to Snowden's whistleblowing, but the content of the documents also upheld assurances that, at that time, Tor could not be cracked [1].

In 2021, when this work is being published, Tor has around 7000 volunteer-run relays and millions of users worldwide [5].

1.2   Design Goals

In their design published in [2], Dingledine et al. state that the main goal of Tor is to prevent attackers from linking communication partners, or from linking multiple communications from or to a single user. However, to achieve this goal, the designers had to consider several key features that Tor should meet:

---

[1]We will talk about them in Section 1.5.

- *Deployability* – The design must not be expensive to run, as it must be usable in real-world conditions. The relay operators must not be required to make excessive contributions, such as high bandwidth availability. Furthermore, Tor should not require them to state their identity or place a heavy liability burden on them (for example, by allowing attackers to use Tor for illegal activities).

- *Usability* – As the key part of staying anonymous is to blend in with the crowd, Tor must be as easy to use as possible, because the more complicated it gets, the fewer people will be willing to use it. Therefore, the user-friendliness of Tor is not just a thing of comfort but directly affects its safety. To be close to all users, Tor should be implemented on all operating systems, so that the users would not be forced to change their habits to stay anonymous.

- *Flexibility* – The protocol should be flexible and well-specified so that Tor can potentially serve as a testbed for future research.

- *Simple design* – The design of the Tor protocol must be kept as simple as possible to maintain readability. Tor aims to deploy a simple and stable system without the need to implement extra techniques that are unproven.

Because Tor's authors prefer a simple and understandable design, from the beginning, they rejected several goals that Tor could have served, but either they are solved elsewhere, or a solution has not yet been found for the problem. Therefore, Tor would be:

- *Not peer-to-peer* – Although for others the approach of a completely decentralized network with thousands of short-lived servers might be appealing, it has still open problems that need to be addressed.

- *Not secure against end-to-end attacks* – Tor's authors do not claim that Tor is completely safe from end-to-end timing or intersection attacks. Note this point, because, in this work, we are the adversary looking at both ends of the Tor connection.

- *Not steganographic* – It is not a goal of Tor to hide that the user is connected to the network.

1.3 Onion Routing

The Onion Routing is essentially the principle on which Tor is built. Instead of connecting to the server directly, the client who wants to access the

server anonymously connects to a series of randomly selected *Onion Routers* (abbr. OR) which redirect the traffic between him and the server. We sometimes refer to the OR as to a *relay* or a *node*.

As described in [2], first, the client downloads a list of trustworthy Onion Routers from a trusted *Directory server*. He selects one of the *Guard relays* to be the first hop in the path between him and the server. Then he negotiates a cryptographic symmetric key K_1 for communication with this relay using the Diffie-Hellman routine so that the exchanged messages would be encrypted. With this exchange, a new *Circuit* has been established between the Client and the Guard relay.

If there was only one Onion Router between the Server and the Client and this would be malevolent, it could eavesdrop on the whole communication. For this reason, using only one Onion Router is not sufficient and normally, at least three hops are used.

To extend the established circuit to a second Onion Router, called the *Middle Relay*, the Client sends a key negotiation request through the Guard Relay to the Middle Relay in a *telescopic* way. This means that for the Middle Relay, the opposite party of the key negotiation process is, therefore, the Guard Relay and not the Client himself, keeping him anonymous from the Middle Relay's point of view. This way a new symmetric key K_2 between the Client and the Middle Relay is established.

To extend the circuit to a third relay or beyond, the Client proceeds as above, always calling the last relay in the circuit to extend one hop further.

When the building of the circuit is finished, the Client is ready to send messages to the Server through the Tor network. To make the message safe from eavesdropping, the Client encrypts it sequentially with the established cryptographic keys in the following manner:

$$c = E_{K_1}(E_{K_2}(E_{K_3}(msg))) \tag{1.1}$$

where E is a cryptographic encryption function. Notice that by encrypting the message in the above manner, the message is encrypted in multiple layers of encryption, which resembles the layers of an onion.

When such an encrypted message travels through the Tor network, as depicted in Figure 1.1, each Onion Router peels off a layer of encryption by decrypting it with its key, effectively leaving the bare message exit the *Exit Relay*. The message exiting the Exit Relay has the form of a message that would be normally sent directly from the client. The Exit Relay, therefore, acts as a proxy, communicating with the destination Server on behalf of the Client. If the messages exchanged between the Client and the destination Server are not encrypted, the Exit Relay might see their content. For this reason, even when using Tor, it is still recommended to use an encrypted protocol, such as *HTTPS*, to communicate with the destination Server in order to maintain the privacy of the communicated content.
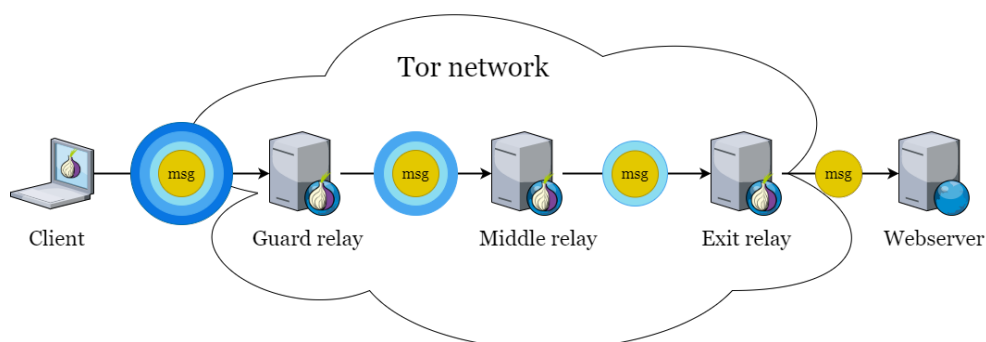
Figure 1.1: Message passed via Tor network

1.4 Tor Browser

Tor Browser, developed as a reaction to the public demand, opened the world of Internet anonymization to the masses. It consists of a bundle of tools, which provide a smooth user experience. The core includes a modified Mozilla Firefox web browser, which is popular among the public and brings a familiar experience to the users. It is, however, extended with TorButton, NoScript, and HTTPS Everywhere Firefox extensions and the TorLauncher, which communicates with the Tor network, negotiates keys, and builds circuits, transparently to the user [6].

As the browsing session starts, TorLauncher starts a Tor background process, which routes the traffic through the Tor Network. When the browsing session terminates, the browser deletes all sensitive data that has been generated, such as cookies and browsing history.

One of the big advantages of Tor Browser is its compatibility; users can run it on Windows, OS X, GNU/Linux, and also on Android, which brought the anonymization services also to, more frequently used, mobile devices. There is also a portable version available, which does not need to be installed and can be run from removable media.

1.5 Onion Services

Onion Services (abbr. OS) are anonymous network services that are exposed over the Tor network as designed in [2]. In contrast to conventional web services, such as websites, Onion Services are private, generally not indexed by search engines, and use a hash of their public key as a domain name, which is hard to read by humans.

When clients connect to such a service, neither do they know the IP address of the server providing the service, nor the server knows the IP address of its

clients. To connect to each other, the Client and the Onion Service need to execute the following steps:

1. OS generates a long-term public key pair to identify his service.

2. OS chooses some ORs as *introduction points* and advertises them on the lookup service, signing the advertisement with his public key.

3. OS builds a circuit to each of his introduction points and tells them to wait for requests from clients.

4. Client learns about the OS from somewhere and retrieves the details about the OS from the lookup service.

5. Client chooses an arbitrary OR as the *rendezvous point* (abbr. RP) for the connection between him and the OS. The RP will act as a middleman for that particular connection. The Client builds a circuit to the RP and gives it a randomly generated *rendezvous cookie*, which will be used for authentication.

6. Client opens an anonymous stream to one of OS's introduction points and gives it a message (encrypted with OS's public key) telling it about himself, his RP and the rendezvous cookie, and the start of a DH handshake. The introduction point sends this info to the OS.

7. OS builds a circuit to Client's RP and sends the rendezvous cookie, the second half of the DH handshake, and a hash of a session key they now share.

8. The RP connects Client's and OS's circuits. Note that RP can recognize neither the Client nor the OS, as it acts as the last hop in both of their circuits.

9. An anonymous stream has been established and Client and OS can communicate, as usual, only a bit slower, as this merged circuit has significantly more hops than a standard one.

1.6 Attacks on Tor

Like many popular services, Tor is a target of frequent attacks. The motivations of attackers are diverse. There are hackers, whose motivation is blackmailing and making a personal profit from data compromising. There are state bodies that eavesdrop on the communication of potential terrorists or criminals. And last, but not least, there are researchers, whose aim is to point to the actual flaws and to provoke the debate on how to make the Internet a safer space.

Most of the attacks on Tor focus on identifying the relationship between the client and the server. This process is called *de-anonymization*. The attacker wants to confirm that the particular connection between the client and the guard relay of Tor and another connection between the exit relay and the destination server belong to the same data flow [7, 8].

According to Yang et al. [9], we can sort the existing de-anonymizing techniques by the activity of the adversary into the following categories:

- *Passive attacks* – A passive adversary does not change the network flow, only taps into it and observes, looking for correlation in statistical properties of the network flows. This is called *traffic analysis*. The attacker tries to measure similarities in the traffic that the client sends and the traffic the server receives.

- *Active attacks* – An active attacker, on the contrary, actively changes the traffic by modifying, deleting, or injecting new data. To achieve this, he uses a compromised OR. Since active attackers are more easily detected, there have been numerous attempts to develop various countermeasures to defend against these threats.

There is one more criterion by which we can differentiate the attacks on Tor; we divide them by the number of measurement points:

- *Single-end* – When an attack is single-end, it means an attacker is eavesdropping only on one end of the network; either between the client and the guard relay or between the exit relay and the destination server.

- *End-to-end* – When the attacker is tapped to both ends of the Tor network (or a multiple of them), this is called an End-to-end attack.

Based on their method and goals, Evers et al. [7] qualify the attacks into the following seven categories:

- *Correlation attacks* – End-to-end, Passive attack,

- *Congestion attacks* – End-to-end, Active attack,

- *Timing attacks* – End-to-end, Active attack,

- *Fingerprinting attacks* – Single-end, Passive attack,

- *Denial of Service attacks* – Single-end, Active attack,

- *Supportive attacks* – Not classified,

- *Revealing hidden services attacks* – Not classified.

In this case, the label "not classified" means that the attack combines both types of techniques.

1.6.1 Correlation Attacks

As Evers et al. [7] explain, correlation attacks are attacks in which the adversary has access to both the guard relay and the exit relay of the circuit between the client and the destination server. The attacker looks for correlation in the statistical properties of flows entering the guard relay and exiting the exit relay because when they correlate, we can say with high probability, that these flows belong to the same connection. And as the client's identity is exposed in the flow between him and the guard relay, so the server's identity is exposed in the flow between it and the exit relay. This means that when the flows are correlated, the attacker is able to identify the communicating pair.

As the correlation itself can be based on various statistical properties, we will now introduce some well-known correlation attacks.

Cell Counter Based Attack

Ling et al. [10] describe an active correlation attack in their paper published in 2012. The attacker needs to manipulate the timing of sending relay cells between the relays and the cell counter on the guard and exit relay. By doing this, he is able to embed his own signal in the communication between relays.

Traffic is sent via Tor cells, which are temporarily stored in a queue, waiting to be flushed to the output buffer and enter the network [11]. By manipulating the number of cells flushed to the output at once, the attacker is able to embed his own signal. For example, when 3 cells are flushed, it means "1"; when 1 cell is flushed, it means "0".

The timing between sending these combined "symbols" needs to be set in a clever way. If the latency is too big, the connection will look suspicious and the user will change to a new circuit. On the other hand, if the latency is too small, signals might merge on the middle relay as a consequence of bandwidth congestion or delay in the network infrastructure. To avoid confusion by these distorted signals, an advanced recovery mechanism with analysis of types of combinations and divisions of these cells was developed.

Thanks to variance in the number of cells used for each symbol and the variable latency, which can be controlled by a pseudo-noise generator, this attack is very hard to detect. On the other hand, when using it, it has near to 100% detection rate and can confirm more than a half of the communication sessions by injecting around 10% malicious onion routes on Tor.

Low-Resource Routing Attack

Bauer et al. [12] described a correlation attack, whose aim is to let the client construct a Tor circuit containing a malicious entry (these days guard) and an exit relay. To achieve this goal, the attacker needs to set up a number of compromised ORs that advertise high bandwidth availability. In 2007, when this paper was written, the trustworthy directory servers did not verify

the advertised bandwidth, therefore an attacker did not have to possess the advertised resources and still could attract clients to use his OR. The more resources malicious OR advertises, the higher the chance of being selected as an entry relay.

If only a single malicious relay is part of the circuit, it can disrupt the path, resulting in the client constructing a new circuit and thus increasing the chance to select 2 malicious relays. This can also be achieved by running a denial-of-service attack on a few key stable entry guards, resulting in a large number of clients having to replace the unusable entry guard with a potential malicious one.

The malicious ORs log enough information to correlate the client request with server responses thanks to a circuit linking algorithm that recognizes a circuit request from a Tor proxy. This is the main difference with other attacks since with this approach, the attacker can compromise the anonymity of the client even before he starts browsing any content.

HTTP-Based Application-Level Attack

In 2011, Wang et al. [13] proposed a HTTP-based application-level attack to identify Tor clients. The attack uses the principle of Man-in-the-middle: The attacker needs to control at least the exit relay, ideally both entry and exit relays. To achieve this, he can use a technique of advertising the high resources available, as described in the previous section.

When the client selects the malicious exit node and tries to connect to the desired server, instead of serving the requested website, the malicious OR serves to the client a forged web page, which initiates multiple malicious connections. This way, the forged web page in the client's browser acts as a beacon, sending a very distinctive traffic pattern.

This pattern can then be detected by the entry relay or a passive observer, watching the link between the client and the entry relay to expose the client's identity.

To avoid this type of attack, it is central to use ciphered connections when visiting websites, which is enforced nowadays by the Tor Browser plugin HTTPS Everywhere.

In 2015, Arp et al. [14] described a similar attack, taking the control of the exit relay out of the equation. Instead, they proposed to use side channels such as banner advertisements or cross-site scripting to deliver malicious content to the client. Then, it is sufficient to only observe the link between the client and the entry relay for the specific pattern generated by the malicious payload.

Bad Apple Attack

The paper, in which this attack is explained has a title *One Bad Apple Spoils the Bunch*, which quite accurately describes what happens in the attack pro-

posed by Le Blond et al. [15].

When a client uses Tor, he builds a set of circuits he uses for a certain amount of time, without changing them or constructing new ones. If a client uses a malicious application, connected to Tor proxy, which, for example, sends out the client's IP address (for instance, a peer-to-peer file-sharing application), the attacker eavesdropping on the circuit's exit relays will be able to correlate traffic from the malicious application with other network traffic from the client.

The Bayesian Traffic Analysis Attack

The Bayesian Traffic Analysis is a probabilistic attack on anonymity mix networks presented in [16] by Troncoso et al. It uses a Markov Chain Monte Carlo inference engine that calculates the probabilities of a selected OR being connected to another OR given an observation of network traces. Finally, everything boils down to calculating an a-posterior distribution $Pr[HS|O, C]$ of a set of hidden state user variables HS given an observation O and a set of constraints C based on the user's choice of mixes to relay messages and the user's behavior.

As discussed in [7], because computing the distribution $Pr[HS|O, C]$ depending on all observations would require enormous computational power, sampling takes its place. Therefore, we can estimate $Pr[HS|O, C]$ with sets $HS_0, ..., HS_n$, which are used to extract the characteristics of the user variables and to infer the distributions that describe events of interest in the system.

This approach enables the attacker to extract information from anonymized traffic traces optimally if he tracks 50 messages from the flow he wants to exploit. In all examples, around 95% of the samples fall into the confidence interval. The results of experiments executed by Murdoch et al. [17] show that when more messages travel through the network, the attacker is less certain about their destination.

The Raptor Attack

There are essentially two ways for attackers to gain access to Tor traffic; either they compromise enough Tor nodes or manipulate the underlying network infrastructure. The Raptor attack, presented in [18] by Sun et al., consists of three individual attacks and exploits the Border Gateway Protocol (abbr. BGP). It assumes that the attackers are powerful enough to control autonomous systems (abbr. AS). There is evidence that intelligence agencies could be such adversaries.

First, Raptor exploits the asymmetric properties of Internet routing. This means that the BGP path from a client to the server can be different from the BGP path from the server to the client. Such asymmetry increases the chances

of the traffic being routed through an AS-level adversary observing at least one direction of both communication endpoints, enabling *asymmetric traffic analysis*. The sequence number of data packets and their acknowledgments can be correlated because the TCP headers of the packets are not encrypted at both ends of the client's circuit, and therefore are visible to the malicious AS.

Second, the attack exploits the natural instability in the Internet routing: BGP paths change over time due to connectivity changes, link failures, or the setup of new relationships between the ASes. These changes increase the chances for the attackers to be in the path of the Tor traffic, which they can observe. Asymmetric traffic analysis is only needed once to correlate the client and the server, which means that the chances of being correlated increase over time.

Third, the attackers can work as a Man-in-the-middle. When they use a BGP interception attack, their malicious AS advertises an IP range that actually does not belong to the AS. When this happens, a portion of traffic headed towards this IP range will be directed through the malicious AS. The attackers can analyze the intercepted traffic and forward it further to the correct destination, without being noticed. This attack applies mainly to links between the client and the guard relay because the IP addresses of the guard relays are well-known.

The DeepCorr Attack

With the rise of Deep Learning algorithms in the last years, the research in correlation techniques naturally turned to use neural networks in this field.

In their paper published in 2018, Nasr et al. [19] introduce a novel approach to correlation itself. Instead of manually engineering the features of the Tor flows, which should be correlated in order to link the flow between the client and guard relay and the flow between the exit relay and the destination server, they simply let a Convolutional Neural Network learn, how the correlation function looks like.

To achieve this goal, they fed the network with data matrices of over 50,000 websites that contain the interpacket delays and packet sizes from the first 300 packets of each flow in both directions. This is a considerably smaller amount of data per flow than a 100 MB file, which was used in the Raptor Attack [18].

The results of the experiment are astonishing. For example, for a False Positive rate (abbr. FPR) of 10^{-3}, DeepCorr offers a True Positive rate (abbr. TPR) rate of 0.8, while the previous systems (Raptor, Mutual Information, Cosine Correlation, Pearson Correlation) offer a rate of TP less than 0.2. It significantly outperforms the prior flow correlation algorithms by very large margins. Importantly, DeepCorr enables the correlation of Tor flows with flow observations much shorter than what was needed by the previous algorithms. This work well demonstrates the escalating threat of flow correlation

attacks on Tor with the rise of advanced learning algorithms and calls for the deployment of effective countermeasures by the Tor community.

CNN

A Convolutional Neural Network (abbr. CNN) is a Deep Learning algorithm specialized in processing data that has a known grid-like topology, for example, image data, signal data, or structured time-series data. Convolutional networks have been very successful in practical applications such as Image Classification, Facial recognition, Recommender Systems, or Advertising. Their name is derived from the mathematical operation called *Convolution*, which is applied to this data.

Each convolutional network has two key components:

- *the feature extraction part*, which consists of several convolution layers, interlieved by the pooling layers,

- *the classification part*, which consists of fully connected (dense) layers.

To comprehend how CNNs work, we first need to define some ground principles, on which the neural networks are built. We will start with the smallest part of a neural network – the *Neuron*.

2.1 Neuron

Neurons are the main building blocks of Deep Learning models. They represent nodes through which data and computations flow. Each neuron works in the following manner:

First, it receives one or more input data $x_1, x_2, ..., x_n$. These data can come either from the raw dataset or from neurons positioned at the previous layer of the neural network. The neuron assigns a weight w_i to each of the incoming data x_i and multiplies its value with it. Weights play an essential role in the Deep Learning process, the model is being trained by adjusting them.

After multiplication, the neuron sums up all weighted values into a single value, which is passed to the *activation function*. The output of the activation
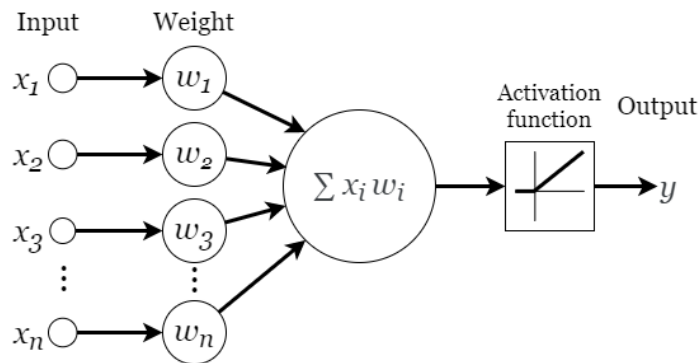
Figure 2.1: Schema of an artificial neuron

function is forwarded to the next layer of neurons in the network. The whole process is illustrated by Figure 2.1 and further discussed in [20].

2.1.1 Activation Function

Deshpande [21] explains that the activation function introduces a nonlinear element to the process, thanks to which the neural network is capable of producing more versatile results. It essentially decides when the neuron is activated and how strong the output of the neuron will be.

In the past, nonlinear functions such as *hyperbolic tangent function* and *sigmoid function* were used, but research has shown, that *ReLu function* works far better from the efficiency point of view, without making a significant difference to the overall accuracy of the network.

ReLu Rectified Linear Units function effectively replaces all negative values of the input with a zero value.

$$f(x) = max(0, x) \tag{2.1}$$

The graph of the ReLu function is a part of Figure 2.1.

2.2 Neural Networks

As stated by Brownlee in [22], neurons are arranged into networks of neurons. A row of neurons is called a *layer* and a network can have a number of them. There are three types of layers: an *input* layer, *hidden* layers and an *output* layer.

Input Layer Input layer is the only exposed part of the network. Its role is to get the values from the dataset and pass them to all neurons in the first hidden layer. Note that the units in the input layer do not perform any calculations, typical for neurons.

Hidden Layers Hidden layers are layers of neurons that are not directly exposed to the input. Their job is to take the input from all neurons in the previous layer, compute the weighted sum depending on their own set of weights, and pass the output of the activation function to all neurons in the next layer.

Output Layer The last layer of the network contains the output neurons, or sometimes only a single output neuron. This layer is responsible for outputting a result value or a vector of values that correspond to the format required for the problem solved by the network.

The choice of the activation function in the output layer is strongly dependent on the type of problem being solved. For example:

- A regression problem may have a single output neuron without activation function.

- A binary classification problem, where the object either belongs to the class (class 1) or does not (class 0), may have also just a single output neuron and use a *sigmoid activation function* to output a value between 0 and 1 to represent the probability of an object belonging to class 1. This can be turned into a crisp class value by using a threshold of 0.5 and snapping values less than the threshold to 0, otherwise to 1.

- A multiclass classification problem may have one output neuron for each class. In this case, we use a *softmax activation function* to output the probability of the network predicting each of the classes. Selecting the output with the highest probability can be used to produce a crisp class classification value.

A *Feed-Forward Neural Network* (abbr. FFNN) is a type of neural network, in which the connections between neurons do not form a circle, meaning that the flow of data always moves forward in the layers and never gets back as feedback. An example of an FFNN is a *Multilayer Perceptron* (abbr. MLP). In MLP, each neuron in one layer has directed connections to all neurons of the subsequent layer. The universal approximation theorem [23] states that every continuous function that maps intervals of real numbers to some output interval of real numbers can be approximated arbitrarily closely by an MLP with just one hidden layer. This is a piece of crucial information to motivate us to use MLPs.

2.2.1 Training

When the input data is prepared in the format the network understands, we can feed it to the input layer. The network processes the input upward activating neurons as it goes to finally produce an output value. This is called a *forward pass.*

MLPs use a variety of learning techniques, with the most popular being *back-propagation*. Using this technique, the output values from the MLP are compared with the correct answers from the training set to compute the value of *Loss function* $\mathcal{L}$ (also called error function). The loss is then propagated back through the network, one layer at a time, and the weights are updated according to the *Stochastic Gradient Descent* (abbr. SGD), calculated from the loss.

Stochastic Gradient Descent   In his article [24], Srinivasan describes the gradient descent as an iterative algorithm that starts from a random point on a function and travels down its slope in steps until it reaches the function's minimum, which is our objective to find. To do this with the whole dataset at once, the computer consumes an enormous amount of time and resources, which is why we need to take steps towards speeding it up. Using a stochastic version of this algorithm helps greatly; instead of using all samples for the calculation, we randomly choose a single one and calculate the SGD according to it.

The aforementioned process of training the weights and comparing them is repeated for all samples in the training dataset. One round of updating the network weights for the entire training dataset is called the *Epoch*. The weights, however, do not need to be updated after each sample, as this would result in a very unstable network. Instead, we can count a sum of losses for a *Batch* of samples and update the weights afterward [22].

We can also control the amount by which the weights can be updated by the *learning rate*, which stands as a multiplier to the result of the SGD. It usually has values of $0.1, 0.01, 0.001$ and smaller. The learning rate is one of the *hyperparameters* of the model, which are manually configurable by the programmer.

2.2.2   Prediction

Once a neural network has been trained, it can be used to make predictions on the validation or test dataset. The network topology and the set of weights are the only things we need to save from the model. To predict the label of the sample from the test dataset, we feed it to the input layer and perform a single forward pass through the network, resulting in a prediction.

2.3   CNN vs. Feed-Forward Neural Networks

In [25], Saha argues that an image, which is a typical input of the CNN, is only a matrix of pixels, and thus would be tempting to flatten it into a single vector of pixel values and feed it to a Multi-Level Perceptron for classification. This

would not work very well, because opposing to the standard features fed to MLP, the features of an image, such as edges, lines, etc., can not be represented in a single pixel. These features are called *spatial* and are only detectable by *filters* that comprehend multiple adjacent pixels. Thus, to extract these features from an input image or other type of matrix data, we need a feature extraction part of the CNN.

2.4 Convolution

Convolution is one of the main building blocks of a CNN. In the context of a CNN, convolution is a linear operation that involves the multiplication of a set of weights with the portion of the input data. Given that the technique was designed for 2D input, the multiplication is performed between an array of input data I and a 2D array of weights K, called a *filter* or a *kernel* (both terms represent the same).

The filter is always smaller than the input data and the type of multiplication applied between a filter-sized portion of the input and the filter itself is a scalar product. Using a filter smaller than the input has its point; it allows the same filter to be applied multiple times at different points of the same input as shown in Figure 2.2. Specifically, the filter window slides over the input data from left to right, top to bottom [26].
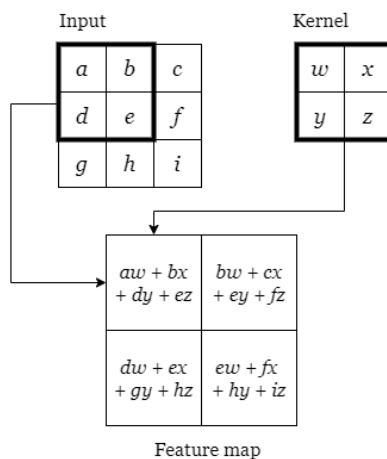

Figure 2.2: An example of 2D valid convolution

Mathematically, we can describe this process as positioning the kernel K of dimensions m x n at coordinates i, j of the input I and calculating the scalar product of underlying values.

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \qquad (2.2)$$

By changing the coordinates i, j on which we apply the kernel and calculating the respective value of the convolution, we build the output of this operation – a 2D array called a *feature map*. To clarify the Equation 2.2, it describes an operation, where the filter is flipped prior to being applied to the input. Technically, the convolution as described for use in CNN is an operation called *cross-correlation*. Nevertheless, when used in CNN, it produces the same set of weights, so we will refer to it as a convolution [27].

Goodfellow et al. [27] explain, that using convolution improves the machine learning process with the following three ideas:

Sparse Interactions   In traditional neural networks, matrix multiplication is used to calculate the weights of the parameters. This would in our case mean that every pixel of an input image would be multiplied with every pixel, causing large demands on the network, while providing no sufficient results. For this reason, CNNs usually use *sparse interactions*, meaning that each pixel interacts only with a few adjacent pixels. This is done thanks to using kernels that limit the interaction of the input pixels only to those within its window. This means that we need to store fewer parameters, which both reduces the memory requirements and also improves the network's statistical efficiency. In deep convolutional networks, the output from one convolution usually represents the input of the next one. Thanks to this, the units contributing to small-scaled kernels can still indirectly interact with a larger portion of the input, which in turn means that the network is able to describe complicated features by connecting many simple features from previous convolutional layers.

Parameter Sharing   In traditional neural networks, each element of the weight matrix is used only once to calculate the output of the layer it is in. It is multiplied by one element of the input and then never revisited. This has quite a big impact on the number of parameters learned, thus on the memory requirements of the model. When using the convolutional neural network, the weight matrix is represented by a filter, meaning that each member of the filter is used at all locations of the input. This means that rather than learning a separate set of parameters for every location, we only learn one set applied everywhere on the input. This reduces the hardware requirements drastically.

Equivariance   If a function is equivariant, it means that if the input changes, the output changes in the same way. Convolution has this property regarding the translation of the input. When we take images as an example, each feature map represents the locations of certain properties on the input image. If we move the object on the image, its representation will move in the feature map as well.

2.4.1 Filters

As the motivation to use convolution on the input is clear, we will define the properties of the filters that are used in this operation. These are also *hyperparameters*, as follows.

Dimensions: They essentially define the width and height the filter window has. To correctly determine the size of the filter, we need to think about how big or small will the features we want to capture be on the input image. As Pandey explains in [28], we can divide the kernel sizes into smaller and larger ones. The smaller have dimensions up to 4 x 4, whereas the larger have dimensions of 5 x 5 and more. These, however, consume such a long time to train that most of the programmers use the smaller ones. To use a kernel of size 1 x 1 is pointless since it does not generalize any features from adjacent pixels. Odd-sized filters symmetrically divide the previous layer pixels around the output pixel. When using an even-sized filter, this symmetry is not present, which can cause distortions across the layers. For this reason, using kernels of size 2 x 2 or 4 x 4 is not recommended. This explains why the kernel size of 3 x 3 is the most popular option for image classification.

Stride: Stride is the size of the step the convolution filter moves each time. It also has 2 dimensions, as we can define the steps along both image axes. A stride size is usually 1, meaning the filter slides pixel by pixel. By increasing the stride size, the filter is sliding over the input with a larger interval and thus has less overlap between the elements. This also means that the bigger the dimensions of the stride are, the smaller the resulting feature map will be.

Padding: When moving the filter window above the input image, not all pixels participate equally in outputting features. Take, for instance, the pixels in the corners of the input image; each of them participates only once. Pixels inside the image, on the contrary, participate several times, as the sliding filter includes them multiple times. When we use no padding, we call it *valid padding*. If we, however, want to include each pixel with equal weight, we can pad up the whole input with border pixels, filled with zero value. This means, that these artificial pixels will not participate in the scalar multiplication, but thanks to extending the input with this padding, each pixel on the border will have the same weight as pixels inside the image. This is, why this padding is called, the *same padding*.

As these three properties are tightly connected, they directly influence the dimensions of the output. To calculate the spatial size of the output, we use the following formula:

$$X = (W - F + 2P)/S + 1 \tag{2.3}$$

where X is the size of output dimension, W is the size of the dimension of the input, F is the size of the dimension of the filter, P is the amount of the zero-padding used on the border and S is the size of the dimension of the stride, as stated by Li et al. in [29].

Number of Filters: As each filter can learn to recognize only one pattern (such as vertical edges, blobs of color, etc.), we need multiple filters to comprehend complex features to accurately classify the input. The number of filters, or sometimes called the *depth* of the convolutional layer is essential to determine the extent to which the network is able to learn. Recall that by applying a filter to an input, we produce a feature map for that particular filter. This means that for every filter we use, a new feature map will be produced. To determine the number of filters to be used, one has to experiment, because there is still no clear relation between the complexity of the input picture and the number of filters used.

2.4.2   Nonlinear Element - ReLu

After applying a convolution resulting in a number of feature maps, it is a convention to apply a non-linear layer (*activation layer*) immediately afterward. The reason for this is to introduce an element of nonlinearity to a system that basically has just been computing linear operations during the convolutions. As already stated in Section 2.1.1, the ReLu function is the most popular nowadays.

2.5   Pooling

After using the activation function, we want to emphasize the features on the feature map so that when we receive a sample that is noisy, the features on it will still be detected. This is called *invariance to translation*, which means that if we translate the input (alter its value) by a small amount, the values of most of the pooled outputs do not change.

   The pooling layer works similarly to the convolution layer. We slide a kernel window of certain dimensions over the feature map and based on the chosen function, we calculate the output value for that particular group of pixels (a portion of input). For the pooling function, we can choose:

Average Pooling   Average Pooling operation returns the average of all values within a rectangular neighborhood, covered by the kernel window. By doing this, it reduces the dimensionality of the input and suppresses the noise.

Max Pooling   Max Pooling operation reports the maximum output within a rectangular neighborhood, covered by the kernel window. As it takes only

the maximum value, it discards the noise activations altogether, meaning it performs denoising along with dimensionality reduction. Hence, we can say that Max Pooling performs a lot better than Average Pooling.

Goodfellow et al. [27] explain that for many tasks, pooling is a crucial operation for handling inputs of different sizes. When classifying images, for example, we need to feed a fixed-size input to the classification part of the network. To make this work, we use the pooling function, varying the size of an offset between the pooling regions, so that the classification part always receives the same number of summary statistics regardless of the input size. For example, the last pooling layer can be defined to return four sets of summary statistics, one for each quadrant of an image, regardless of the image size.

The convolutional layer and the pooling layer together form the i-th layer of a CNN. The number of these layers used varies depending on the complexity of the input images. The more complex the input images are, the more layers of convolution and pooling need to be used, as described in [25]. This, however, comes with the price; adding a new layer significantly increases the hardware and time demands of the CNN.

After going through the above process of *feature extraction*, we have successfully enabled the model to extract the features from the input data. Now it is time to flatten the output of the final pooling layer and feed it to a regular neural network for classification purposes.

2.6 The Classification Part – Fully Connected Layer

Recall the contents of the Section 2.3, in which we compared CNN vs. Feed-Forward Neural Networks. We argued that the spatial features are noncomprehensible for the standard neural network, which made us create a CNN to do the job. The fact is that a CNN includes a Feed-Forward Neural Network as its classification part – the Fully Connected Layer.

After applying the last pooling layer, we need to convert multiple pooled feature maps into a form a Multi-Layer Perceptron understands – a vector of values. To accomplish this, we use a *flatten function*, which merges all values from the feature maps into a single vector, fed to the MLP. The classification process then continues exactly as described prior in Sections 2.1 – 2.2.2.

Network Flow Monitoring

Network monitoring has been here since the beginning of the Internet. Once the devices were connected via networks, there was a demand for remote monitoring of their activities, as checking them all in person became unbearable. To do so, there were numerous approaches proposed and developed throughout the years, each of them serving a different purpose. Generally, we can classify them into the following categories, as stated in [30] by Hofstede et al.:

- *Active* – With an active approach, we inject our own traffic into a network to perform different types of measurements. This includes using tools like Ping or Traceroute.

- *Passive* – With a passive approach, we observe the existing traffic as it passes by the monitoring point and therefore observe the traffic generated by users without interfering.

Two of the well-known passive approaches are Packet-based and Flow-based analysis. In the following sections, we will briefly introduce the history, the architecture, and finally, the motivation that drives us to prefer the Flow-based analysis to Packet-based analysis.

3.1   History

In his article [31], Edwards explains that in the early days of the Internet, there were several working groups working out the solution for the problem, but none of those became a universal standard. It was not until 1988 that the monitoring approaches were standardized by the *Simple Network Management Protocol* (abbr. SNMP). This protocol effectively carried network management information between the devices, thus providing a language, in which different devices could talk to each other.

Not much later, in 1991, the aggregation of packets into flows by means of packet header information was described in [32] for accounting purposes. At

that time, however, the common belief was that the Internet should be free, meaning that no means of traffic capturing, potentially leading to accounting, monitoring, etc. should occur. For this reason, the idea was abandoned for a few years.

Although from 1995 to 2000, the new Realtime Traffic Flow Measurement (abbr. RTFM) Work Group worked on the research and modeling of an improved traffic flow model, due to vendors' lack of interest, no flow export standard resulted.

Parallel to RTFM, Cisco invented its flow export technology named *NetFlow*, which originated as a secondary product in their new switching technology, as described in [30] by Hofstede et al. In flow-based switching, the switching decision is made only for the first packet of the flow, instead of all packets, as was the practice before. The flow information is maintained in a *flow cache* and forwarding decisions for all subsequent packets in the flow are made based on the data stored in the cache. To export these data was only a small, last step towards flow export technology. NetFlow was patented by Cisco in 1996. The first wide adopted version was NetFlow v5, published in 2002. Although Cisco did not publish any official documentation, the data format of the NetFlow was publicly available, making NetFlow the first choice for the variety of vendors. In 2004, NetFlow v5 was superseded by v9, which was also selected as the basis of the new *IPFIX* Protocol [33], which is a standard of flow-based monitoring nowadays.

3.2   Architecture

According to Hofstede et al. [30], the architecture of a typical flow monitoring system consists of the following stages:

Packet Observation   In this stage, the packets are captured and preprocessed at the *Observation Point*, which can be a passive monitoring probe or an active device capable of traffic monitoring.

Flow Metering and Export   This stage consists of two processes, namely a *Metering Process* and an *Exporting Process*. The Metering Processes task is to aggregate packets into flows, which are defined as "sets of IP packets passing an observation point in the network during a certain time interval, such that all packets belonging to a particular flow have a set of common properties" [34]. When the flow meets the conditions under which it is considered terminated, a flow record is exported by the Exporting Process, which effectively means, that the Exporting Process places all gathered data about the flow into a datagram satisfying the requirements of the flow export protocol. The exported data include both characteristic properties of a flow, such as IP addresses and port numbers, and also measured properties, for example, packet and byte counts.

Data Collection   Tasks in the data collection stage include reception, storage, and preprocessing of the flow data generated by the previous stage. Common preprocessing operations include aggregation, filtering, compression, and generation of summaries.

Data Analysis   The last stage of the process depends on the deployment of the system. In research, data analysis is often done manually in order to find relevant features. On the other hand, in operational deployments, the analysis functions are often integrated into the Data Collection stage.

Common analysis functions include correlation and aggregation, traffic profiling, classification, characterization, anomaly and intrusion detection, and search of archival data for forensic or other research purposes.

This conceptual design is often transformed into devices, which include multiple stages, closely interconnected: A *Flow Exporter* is a single device containing both Packet Observation and Flow Metering and Export stages. If it resides in a standalone device, we refer to it as a *flow probe*. On the other hand, when collecting data from multiple flow exporters, we refer to such appliance as a *flow collector*, as depicted in Figure 3.1.
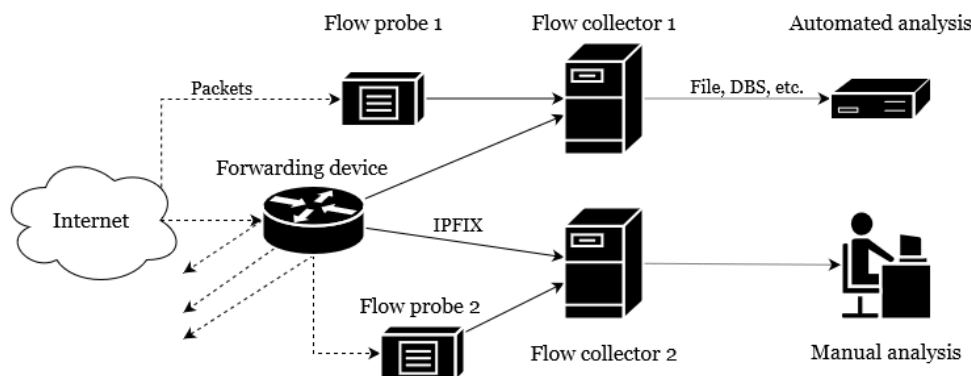
Figure 3.1: Schema of network monitoring parts [30]

3.3   Packet-Based vs. Flow-Based Analysis

Although flow-based analysis solutions are great for monitoring purposes, there still are areas where packet capture and analysis are required.

Packet analysis is usually associated with *SPAN ports*, which are available on most managed network switches. They use *Port Mirroring* to copy each packet passing through the device to the network monitoring connection on another switch port. The monitoring personnel or researchers then can inspect the captured connections packet-by-packet in order to determine flaws in the

communication of a particular client or application. *Deep packet inspection* (abbr. DPI) applies to technologies that use packets as a data source and then extract metadata such as application or website names. The packet-based analysis is therefore an in-depth analysis, contrary to flow-based analysis, which provides only summarised data. One of the main reasons, why we do not use the packet-based analysis all the time, are resources. To store packet data in high-speed networks with speed rates above $100\,\mathrm{Gb/s}$, packet capture requires expensive hardware and substantial infrastructure for storage and analysis. Using the flow-based analysis, we can reduce the volume of stored data to 1/2000 of the original volume, as stated in [30].

Therefore, we usually use flow-based analysis for standard monitoring and enable packet capture on-demand, when there is a reason for a deeper investigation of network traffic.

As the IPFIX protocol and its implementations advance, there are fewer reasons to turn to packet analysis. Advanced flow exporters provide, among other things, data from Layer 7 – the Application Layer, which is the data we would normally use packet capture for. As Minarik states in [35], by enriching traditional flow statistics with application-layer visibility, flow analysis can handle up to 95% of troubleshooting tasks.

Correlation Attack using Deep Learning

As we introduced all main underlying technologies, we can connect them in order to execute a successful de-anonymization attack. First, we will design a laboratory environment, in which the attack takes place. Next, we will discuss the selection of targets in order to analyze a typical Tor user's behavior. Afterward, we will collect the data in our laboratory and evaluate them using a Deep Learning algorithm, and finally, we will present the results of our experiments.

4.1   Lab Design

In the experiments with DeepCorr done by Nasr et al. [19], the researchers used the Tor network to tunnel the traffic between the client computer and a SOCKS proxy server, which they set up. They captured the encrypted Tor traffic exiting their client, as well as traffic exiting the SOCKS proxy server with `tcpdump` tool. By using the SOCKS proxy server as their endpoint, the captures of traffic from the client to the Tor guard relay effectively include also the tunnel established between the client and the proxy. On the other side, the samples captured on the proxy contain pure data flow between the client and the destination web server.

We aim to get closer to the real-world scenario, as depicted in Figure 4.1. Instead of the SOCKS proxy server, we use our own Tor exit relay as a capture endpoint, which we set up to allow only web traffic. This way, the samples we collect will contain the natural noise of the running network, which our capture tools will have to deal with.
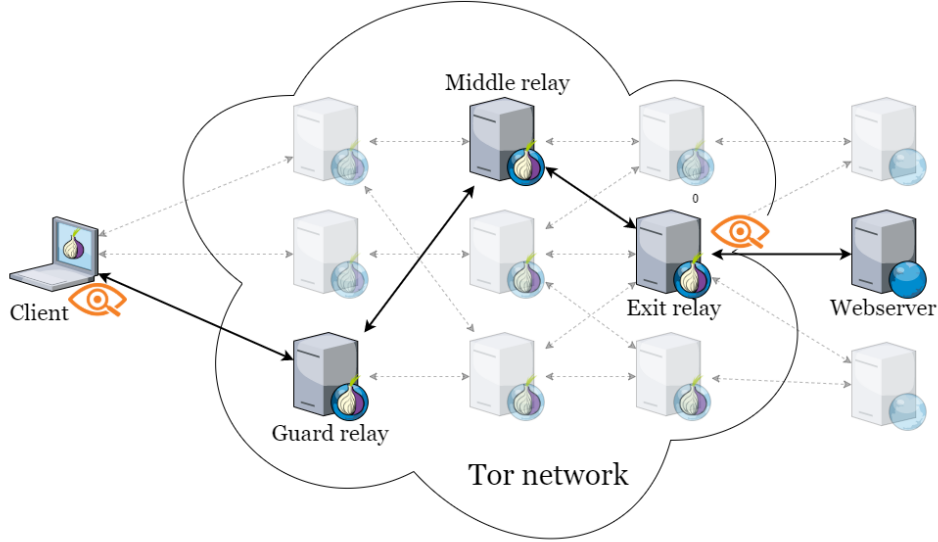
Figure 4.1: Schema of a laboratory design

4.1.1   Hardware and Connectivity

For the client computer, we decided to use a VMWare virtual machine, running Linux Mint and a standard Tor browser with the latest version (10.0.15) with an Internet uplink of 15 Mb/s to simulate a standard client approach as accurately as possible.

For Tor exit relay, we rented a VPS, running a Centos 8 operating system, with a 2 TB traffic quota per month. Knowing this, we needed to limit the daily bandwidth consumption to 50 GB. To distribute this bandwidth evenly throughout the day, we limited the upload and download speed to 1 Mb/s. This also introduced a necessary latency into our experiment. As our experiment aims at web browsing, we decided to allow only web traffic at the exit relay, resulting in filtering only the traffic directed to ports 80, 8080, and 443.

As for the Deep Learning process, we used Google's Colab environment, which provided us with enough computing power to run the following experiments smoothly.

4.2   Selection of Targets and Evaluation Method

Nasr et al. [19] selected the top 50,000 Alexa websites as their target. We agree with this approach because when the Deep Learning model learns on most common cases, it will be able to generalize better as if it would learn

on a few very specific websites. For our experiment, we used the first 15,000 websites of Alexa's Top list, usually in batches of 5000. The main difference of our experiment lies in the amount of data used for training and detection; while Nasr et al. [19] used the first 300 packets in each direction to train their DeepCorr CNN, we plan to use aggregated data from only the first 100 packets in both directions, reducing the data size to 1/3.

As our experiment is heavily inspired by the work of Nasr et al. [19], we also use the Convolutional Neural Network as a Deep Learning algorithm. However, our data structure and length of samples differ from data in [19] (recall Section 1.6.1) and thus we altered the data preprocessing phase and model's hyperparameters to suit our goal.

4.3 Data Collection

4.3.1 Tools for Capture Automation

To automatize the data collection as much as possible, we choose the following well-known tools:

TShark

TShark is a network protocol analyzer. As stated in [36], it lets the user capture packet data from a live network, or read packets from a previously saved capture file, either printing a decoded form of those packets to the standard output or writing the packets to a file. TShark's native capture file format is pcapng format, which is also the format used by Wireshark and various other tools.

In our experimental scenario, TShark was used on the client computer and the exit relay simultaneously. To filter out out-of-scope traffic, a capture filter was applied. On the client computer, only packets flowing to and from Tor guard relay IP Address were filtered out. This IP Address was provided by the automation script we will introduce later. All other traffic, including the Tor circuit establishing via directory servers, was excluded. On the exit relay, only traffic flowing to and from ports 80, 8080, and 443 was captured. This ensured that only traffic exiting the Tor network was trapped since our exit relay was acting as a guard and middle relay for different Tor circuits.

Selenium WebDriver, Stem

Selenium is an umbrella project encapsulating a variety of tools and libraries enabling web browser automation. Selenium specifically provides an infrastructure for the W3C WebDriver specification – a platform and language-neutral coding interface compatible with all major web browsers [37].

31

In our experiment, we used a tbselenium flavor of this framework, which automates the opening of website batches in the Tor browser. This was combined with Stem, which is a Python controller library for Tor. Using these two together, we were able to ensure that a new circuit was created for every website and that our exit relay was enforced in every created circuit on our client computer.

4.3.2 Data Collection Process

The correct data collection process is a foundation for successful research. That is why we tried to make the whole process as rigorous as possible.

First, we load the first N domains from the list of Alexa's Top Million sites and distribute them evenly through our processor pool. Each kernel takes its part from the pool and starts the browsing process:

In the beginning, we create a new Tor process, controlled by Stem. At this point, we configure the Tor process to strictly use our Tor exit relay, so that we would be able to capture the outgoing traffic at that point. Next, we start capturing the traffic with Tshark on both the client computer and our Tor exit relay. We set up Tshark to capture only for 20 seconds, which should be far more for the first 100 packets than we need.

Then, we initiate the Tor Browser via Selenium WebDriver to load the website of the domain from the list. We instruct the WebDriver to wait to finish the page loading for max. 20 seconds. Afterward, the browser is forcefully terminated. If the website is loaded prior to this timeout, we wait only 2 seconds for a fully loaded website.

When both captures are done, we copy the capture file from the Tor exit node to our client's local storage, so that we have both capture files in the same place ready for further processing. This results in a pair of files named by convention `domain.com-DATE-TIME-in.pcapng` for traffic between the client computer and Tor guard relay, captured on the client computer, and `domain.com-DATE-TIME-out.pcapng` for the traffic between Tor exit relay and the destination server, captured on the Tor exit relay.

To variate the collected dataset, we captured different numbers of domains, mostly in batches of 5,000, in different parts of the day on different days of the week.

4.3.3 Tools for Data Preprocessing

To extract relevant statistical data from the raw capture files provided by Tshark, we decided to use NEMEA (Network Measurements Analysis), which is, according to Cejka et al. [38], a streamwise, 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). Originally, NEMEA has been developed for the purposes of the
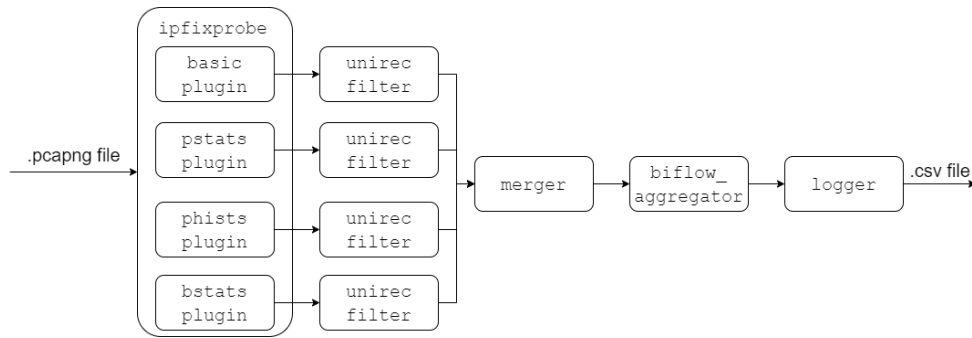
Figure 4.2: NEMEA modules connection

Czech National Research and Education Network operator. Therefore, it is focused on handling high-speed network traffic on links working at 100 Gb/s.

In our experiment, we use modules `ipfixprobe`, `unirecfilter`, `merger`, `biflow_aggregator` and `logger` pipelined via temporary files. Output from one module therefore acts as an input for the next one.

We use `ipfixprobe` with its plugins to extract significant aggregated data from raw packet data. This includes information about the first 100 packets (their length, direction, and timestamp), histograms of packet sizes and inter-packet delays, and statistics about data bursts, covering their duration and size. The output of these plugins is then filtered via `unirecfilter` module, which filters out only flows we are interested in. Afterward, outputs from all plugins are merged to a single file in the `merger` module and finally aggregated into a single bi-directional flow, which is exported via `logger` into a CSV file.

4.3.4 Preprocessing

For further data analysis, it is crucial to extract relevant statistical data from the raw capture files. For this purpose, we built a python3 script `preprocess.py`, which connects all necessary NEMEA modules as displayed on Figure 4.2. The whole process is divided into 6 phases.

Phase 0 - Data Pairing

For each `*-in.pcapng` file in our data collection directory, there must be an `*-out.pcapng` file present, and both of them need to have at least 20 kB in size. If this condition is not satisfied, the sample is thrown away. We build a list of such tuples for further parallelization.

Then, we feed this list of tuples into a pool of processes to make the whole preprocessing quicker.

33

Phase 1 - Destination IP addresses

As we were capturing several traffic flows on our Tor exit relay simultaneously, naturally, there was data noise in our samples, consisting of web traffic that had nothing to do with our measured web browsing. To filter out only the web flows from the measured website, we use the `unirecfilter` module, with the filter set up to filter out only IP addresses that are the destination IPs for TLS handshake protocol. Since this protocol provides a *Server Name Indication* (TLS_SNI) which consists of a domain name, we can use this field to filter only those TLS_SNI's that contain the name of the domain we were actually browsing. The result of this phase is a list of IP addresses, which will be used in later phases.

This phase is applied only on `*-out.pcapng` samples, which were captured on our exit relay.

Phase 2 - Get Plugin Data

In this phase, we run `ipfixprobe` module on each sample. As mentioned above, this module's task, with its plugins, is to extract specific aggregated data from the provided sample.

Plugin `basic` extracts basic flow data such as:

- SRC_MAC – source MAC address,

- DST_MAC – destination MAC address,

- SRC_IP – source IP address,

- DST_IP – destination IP address,

- TIME_FIRST – first timestamp,

- many others which are, for our purpose, irrelevant [39].

We use the field SRC_IP as a key to merge outputs from other plugins into a single row. Field DST_IP is used to filter out only the flows from and to the momentarily browsed web domain. Field TIME_FIRST, which represents the timestamp of the first packet in the flow, is used to calculate time differences in the following time-based fields since we are interested in the relative times opposing to absolute time stamps.

Plugin `pstats` gathers statistics for the first N (30 by default) packets in the flow record. We use this plugin with $N = 100$ to provide enough data. The plugin outputs the following fields:

- PPI_PKT_LENGTHS – array of sizes of the first N packets,

- PPI_PKT_TIMES – array of timestamps of the first N packets,

- PPI_PKT_DIRECTIONS – array of directions of the first N packets,

- PPI_PKT_FLAGS – array of TCP flags for each packet [39].

We use the first 3 of these fields to accurately describe the properties of each flow. The field PPI_PKT_TIMES will be later normalized into an array of interpacket delays. Directions in PPI_PKT_DIRECTIONS are represented as values 1 for outgoing traffic and -1 for the incoming traffic.

Plugin `phists` exports the histograms of packet sizes and interpacket delays for each direction. The histogram bins are scaled logarithmically and are shown in the following table:

Bin Number	Size	Interpacket Delay
1	0–15 B	0–15 ms
2	16–31 B	16–31 ms
3	32–63 B	32–63 ms
4	64–127 B	64–127 ms
5	128–255 B	128–255 ms
6	256–511 B	256–511 ms
7	512–1023 B	512–1023 ms
8	$\geq$ 1024 B	$\geq$ 1024 ms

These histograms are represented as the following arrays:

- S_PHISTS_IPT – histogram of interpacket delays (SRC $\rightarrow$ DST),

- S_PHISTS_SIZES – histogram of packet sizes (SRC $\rightarrow$ DST),

- D_PHISTS_IPT – histogram of interpacket delays (DST $\rightarrow$ SRC),

- D_PHISTS_SIZES – histogram of packet sizes (DST $\rightarrow$ SRC) [39].

Note that these histograms are based on data from the entire duration of the capture, not only from the first 100 packets.

Phases 3 and 4 - Merge and Aggregate

In these phases, the output from each plugin is taken and merged into a single row in our data table. When browsing webpages, the webpage naturally loads many of its contents asynchronously to speed up the user experience, which means that multiple output flows represent the same webpage. For our purpose, we merge the data from these individual flows into a single, aggregated flow. First, we append the data of each subsequent flow at the end of the first flow. Next, we order all data in the row according to the corresponding timestamps. Histograms are, of course, summed together.

Phase 5 - Export

Finally, the aggregated flows from the client computer and exit relay are serialized and saved as a Python dictionary entry in a Python pickle format. These files are later directly called by the Tensorflow library, which builds a dataset for CNN from them.

4.4   Data Evaluation in CNN

4.4.1   TensorFlow

TensorFlow is an open-source software library for numerical computation and large-scale machine learning using data-flow graphs. It bundles together a number of machine learning and Deep Learning models and algorithms. It uses Python to provide a convenient front-end API for building applications with the framework while executing the whole process in high-performance C++, which makes it compatible from mobile devices to large, distributed architectures.

The Layers API provides a simple interface for commonly used layers in the Deep Learning models. On top of that sit higher-level APIs, such as Keras and Estimator API, which make training and evaluating distributed models easier.

4.4.2   Feature Extraction

In the original experiment, Nasr et al. [19] used the first 300 packets in each direction as the base data for their classification problem. They represented them as an image with dimensions 8 x 300 whose rows represent:

1. CLIENT: interpacket delays – download,

2. SERVER: interpacket delays – download,

3. CLIENT: interpacket delays – upload,

4. SERVER: interpacket delays – upload,

5. CLIENT: packet sizes – download,

6. SERVER: packet sizes – download,

7. CLIENT: packet sizes – upload,

8. SERVER: packet sizes – upload.

In our experiment, we use only a portion of these data, namely, only the first 100 packets in both directions and histograms from the whole capture (20 seconds long).
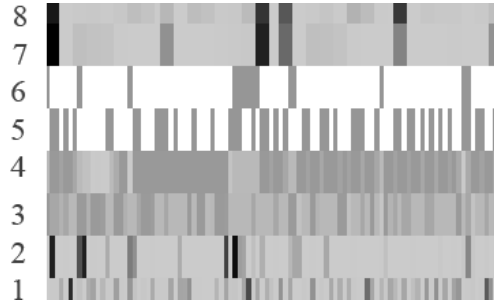
Figure 4.3: Image representation of bi-flow data

We represent these structured data as an image with dimensions (h x w) 8 x 100 whose rows represent:

1. CLIENT: interpacket delays,

2. EXIT_RELAY: interpacket delays,

3. CLIENT: packet sizes,

4. EXIT_RELAY: packet sizes,

5. CLIENT: packet directions,

6. EXIT_RELAY: packet directions,

7. CLIENT: histogram of interpacket delays (download), histogram of packet sizes (download), histogram of interpacket delays (upload), histogram of packet sizes (upload),

8. EXIT_RELAY: histogram of interpacket delays (download), histogram of packet sizes (download), histogram of interpacket delays (upload), histogram of packet sizes (upload).

In the Figure 4.3 an example of such an image is presented. Note that all rows are normalized in stripes of 2 rows to contain only values between 0 and 1. At the beginning of the experiment, we tried to use two more feature rows, namely, data about traffic bursts. These, however, confused the network instead of helping it and therefore were removed.

4.4.3 Creating the Dataset

To train our network, we use a large set of flow pairs that we created over Tor. This includes a large set of associated flow pairs and a large set of non-associated flow pairs.

An associated flow pair $F_{i,j}$, also referred to as a *positive sample*, consists of two segments belonging to the same Tor connection (e.g., i is the ingress

and j is the egress segment of a Tor connection). In other words, i represents the flow data captured on the client computer, while j represents the flow data captured on our Tor exit relay. We label an associated pair with $y_{i,j} = 1$.

On the other hand, each non-associated flow pair also called a *negative sample*, consists of two arbitrary Tor flows that do not belong to the same Tor connection. We label such non-associated pairs with $y_{i,j} = 0$.

For each captured Tor entry flow i we generate N_{neg} negative samples by forming $F_{i,j}$ pairs where j is the exit segment of an arbitrary Tor connection. N_{neg} is a hyperparameter whose value was obtained through experiments. This is identical to the work of Nasr et al. [19].

First, we load the data from Python pickle files produced in Section 4.3.4 Phase 5 - Export. Next, we enter a loop, in which we extract the features from the data as per previous section 4.4.2 and generate N_{neg} negative samples from randomly picked flows. Next, we shuffle the array of indices of the whole dataset and split it by ratio *train_ratio* (usually around 0.8) into two disjunctive arrays: training and testing set. These arrays of indices are later called from the training and prediction process.

4.4.4   Model

For our experiments, we used the model from Nasr et al. [19] and changed the hyperparameters to fit our data structure.

The CNN takes a flow pair $F_{i,j}$ as the input. The model consists of two convolutional layers and three dense layers. The first convolution layer consists of $k_1 = 2000$ kernels[2] each of dimensions 2 x w_1, where w_1 is the hyperparameter, determined experimentally. The reason behind using this layer is to find the potential correlation between two adjacent rows of the input image, which represent the same type of data on both ends of the Tor connection. After this layer, the model uses max pooling to avoid overfitting [27].

The second layer of convolution is designed to combine the extracted features from the first layer into a single feature vector. This means that the CNN is looking for combined patterns in interpacket delays, packet sizes, their directions, and histograms at once. At this layer, the model uses $k_2 = 800$ kernels, each of dimensions 4 x w_2, where w_2 would be experimentally determined.

The output of the second convolution layer is max-pooled and finally flattened and fed to the dense layers which represent a fully connected network. Finally, the output of the network is:

$$p_{i,j} = \Psi(F_{i,j}) \tag{4.1}$$

which is used to decide if the two input flows in $F_{i,j}$ are correlated or not. To normalize the output of the network, we apply a sigmoid function [27] that

---

[2]We did not change this hyperparameter from the original work in [19].

scales the output between zero and one. Therefore, $p_{i,j}$ represents the probability of the flows i and j being associated (correlated), thus being the entry and exit segments of the same Tor connection. The network declares the flows i and j to be correlated if $p_{i,j} > \theta$, where θ is experimentally determined *detection threshold*.

4.4.5 Hyperparameters

We experimented with the setting of the following hyperparameters:

- N_{neg} – number of negative samples for each associated flow,

- k_1 – number of kernels in the first conv. layer,

- w_1 – width of kernels in the first conv. layer,

- m_1 – width of max pooling after the first conv. layer,

- k_2 – number of kernels in the second conv. layer,

- w_2 – width of kernels in the second conv. layer,

- m_2 – width of max pooling after the second conv. layer,

- lr – learning rate,

- dr – dropout.

For the standard training process, we used 4,420 associated flows and experimented with $N_{neg} = \{1, 9, 99, 199\}$ negative samples for each associated flow. Using 199 negative samples produced the best results, although not too far from 99, concerning the fact that 99 negative samples for each associated flow make a dataset half the size of the one with 199 negative samples.

Nasr et al. [19] experimented with learning rates $lr = \{0.001, 0.0001, 0.0005, 0.00005\}$ and got the best results with $lr = 0.0001$. Since the nature of the data in our experiment is similar to data in the original experiment, we decided to fix this value in our experiments to the best measured value by Nasr et al., thus $lr = 0.0001$. The same applies for the dropout value $dr = 0.6$.
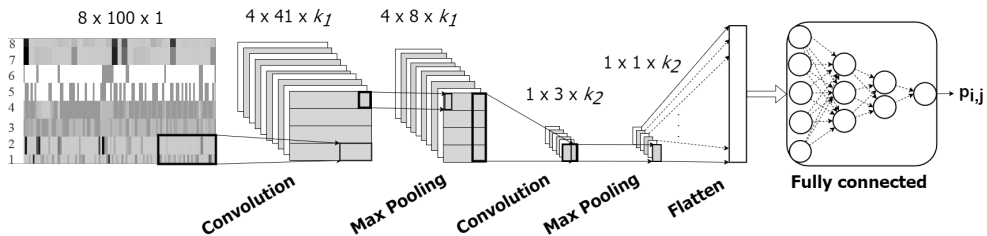

Figure 4.4: The CNN architecture

As the best size for w_1 in original experiment was $w_1 = 30$ for images of width 300, for our images of width $w = 100$, we experimented with $w_1 = \{3, 5, 10, 20\}$. For the first convolutional layer, $w_1 = 20$ yielded the best results. As the height of the convolutional window was statically set to 2, so that the rows of the image would be processed in pairs, we also set the stride for this convolution to 2 x 2. This ensured that the filter would be applied to pairs of input rows, containing the same type of data. Note that the hyperparameters are set in such a fashion that no padding is necessary, thus the layer has the padding set to value *"valid"*.

For the second convolutional layer, $w_2 = 3$ produced the best results. Note that the value 3 is proportionally accurate to the setting from the original experiment done by Nasr et al. ($w_2 = 10$ for $w = 300$), since our width w is three times smaller. In this layer, again, we do not use any padding, since the hyperparameters are set with respect to our data image dimensions.

We also tried different sizes of the max pooling windows m_1, m_2. Although for the first max pooling, the original value of $m_1 = 5$ fitted the best, for the second round, we diverged to the value of $m_2 = 3$.

4.4.6 Training

The training process consists of the main loop, called the *Epoch* as defined in Section 2.2.1. In this loop, we pass the entire training dataset to the CNN. As the dataset might be too big to process at once, we divide it into chunks, called *Batches*. In our experiments, the number of epochs is around 20, while the size of a batch is 512 samples. These sizes are dependent on the hardware used for running the CNN.

After the process enters the Epoch loop, a generator provides it with a Batch – a randomly shuffled list – of input data from Section 4.4.3. This data is then fed to the model, which in turn, returns its prediction for that particular input. From this prediction the *Loss* is computed by the *Loss function*.

For the Loss function, we use the average binary cross-entropy defined as:

$$\mathcal{L} = -\frac{1}{|\mathcal{F}|} \sum_{F_{i,j} \in \mathcal{F}} y_{i,j} \log(\Psi(F_{i,j})) + (1 - y_{i,j}) \log(1 - \Psi(F_{i,j})) \qquad (4.2)$$

where $\mathcal{F}$ represents the current training batch, composed of associated and non-associated flow pairs and $y_{i,j}$ represents the true label of the sample.

In the binary cross-entropy function, as described in [40] by Koech, each predicted class probability is compared to the actual class desired output 0 or 1, and a loss is calculated that penalizes the probability based on how far it is from the actual expected value. The penalty is logarithmic in nature, yielding a large score for large differences close to 1 and a small score for small differences tending to 0. Cross-entropy loss is used to adjust the model weights during the training process. The aim is to minimize the loss, i.e, the smaller the loss, the better the model.

After applying the loss function, we use the gradient tape to automatically retrieve the gradients of the trainable variables with respect to the loss. Subsequently, we use the Adam optimizer [41] to update the weights of the model based on the gradients.

The loss produced in the previous step is added to the sum of losses, effectively counting the overall loss in the epoch. To determine the precision of the model at the current stage, we introduce the *Accuracy* metric, which compares the output $S(\Psi(F_{i,j}))$ of the current learning round with the assigned sample label $y_{i,j}$ for each sample included in the batch, where $S(x)$ represents a sigmoid function. Similarly to the overall loss, the current accuracy is added to the overall accuracy in the epoch.

Finally, the whole inner loop of the epoch, called the *Step* is done and a new batch is generated. The whole inside of the epoch loop can start again until all batches have been consumed.

When all batches have been processed, the Epoch loop ends returning means of loss and accuracy, calculated from their respective sums.

4.4.7 Cross-Validation

The goal of cross-validation is to test the model's ability to predict labels for data that have been not used for training, in order to flag problems like overfitting or selection bias and to give an insight on how the model will generalize to an independent dataset.

The model updated in the latest epoch in the training process is immediately used to cross-validate the small pre-generated portion of the training dataset. Essentially, we run an extra epoch on these data, only this time, we do not update the weights after the prediction is done by the provided model.

Subsequently, the mean of loss from the cross-validation is compared to the lowest loss from the previous epochs. If it is smaller, it means that the current model is performing better than the previous models and so it can be saved on the disk as a temporarily best one.

4.4.8 Testing

After completing the training, the model can be used to predict whether the two presented flows originate from the same connection or not. To test the model extensively, we provide it with a part of the dataset that has never been used for training, therefore the model has never seen these particular flows.

The whole prediction process runs in a similar fashion to the cross-validation. We run only one epoch on the test data and as a result, we receive the means of loss and accuracy, which are the final metrics to determine the precision of the model's predictions.

Results of Experiments

First, we will define what metrics we care about and the reasons why. As a positive sample, as already defined in Section 4.4.3, we will consider samples that contain a pair of flows that belong to the same connection. Such samples will have a label "1". All others will have a label "0" and will be referred to as negative samples.

The standard metric in evaluating the machine learning models is accuracy. This number represents the ratio between correct predictions and the total number of predictions. As long as the dataset is balanced (both classes are represented by an almost equal number of representatives), the accuracy can offer a good insight into the quality of the model. In our case, however, the datasets will be highly imbalanced: as for the practical scenario, there will be one positive sample to be found among thousands of negatives. For this reason, metrics such as True Positive and False Positive rates, precision, and recall will have more predicative value than accuracy.

When considering the practical use of our classifier, we aim for a high number of True Positive (abbr. TP) values, because that means that our classifier predicted the correlation of a particular flow correctly. As for False Positives (abbr. FP), we would like to have the number of them quite low, but in a practical sense, these falsely accused samples can still be later examined by the staff operating the classifier. False Negatives (abbr. FN), however, show the real weakness of the classifier, as this number represents the number of correlated flows our classifier did not detect at all. We aim for this number to be as low as possible, even if that would mean an increase in False Positives. And finally, True Negatives (abbr. TN), the number representing the amount of correctly predicted unrelated flows, is from our point, the least interesting, because we are mainly focused on correctly detecting the positive samples.

To evaluate the quality of our model, presented in the previous chapter, we executed multiple experiments using different datasets, following in the next sections.

5.1   Standard Control Dataset

As a first experiment, we evaluated a dataset captured in the same conditions, with the same Tor exit relay, with the same client and connection speeds as the model was trained on. We used 3,156 positive samples originating from websites with Alexa's rank 5,000 – 10,000, meaning that the control samples are completely disjunctive to the training dataset, which was obtained from pages with rank 1 – 4,999.

We set $N_{neg} = 999$, meaning that the final number of samples tested equals to 3,156,000. The resultant accuracy of 0.999 can look impressive at first sight, but as already mentioned, it does not provide us with any valuable information. For this reason, we need to dive a bit deeper and examine the True Positive, False Positive, and False Negative metrics. To do this, however, we need to know the value of the detection threshold θ (recall Section 4.4.4). This value can be found as an optimal tradeoff between precision and recall. We thus present Figure 5.1 in which we see the aforementioned relationship depending on the variable threshold θ. To comprehend both precision and recall in a single metric, we define the F_β-score, which is defined as follows:

$$F_\beta = \frac{(1 + \beta^2) \, . \, \text{precision} \, . \, \text{recall}}{(\beta^2 \, . \, \text{precision}) + \text{recall}} \tag{5.1}$$

In our case, as already explained, we put more weight on the low number of True Negatives (thus high recall), than lower False Positives (thus we can allow a bit lower precision). To reflect this in the score, we set $\beta = \{2, 3, 4, 5\}$, with $\beta = 4$ producing the best Precision – Recall tradeoff. This means that we will always calculate the F_4 score and in the Precision – Recall plot, we will be looking for such a point, which yields the maximal value of F_4.
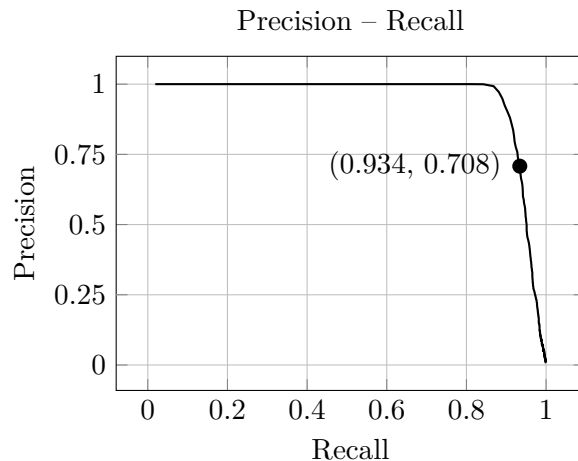

Figure 5.1: Standard control dataset: Precision – Recall

If we aimed at higher betas, the number of False Positives would outgrow the number of True Positives, which would be unacceptable. On the other hand, with lower betas, the number of True Negatives would rise, which is in direct conflict with our goal to reduce it as much as possible.

The best F_4 score achieved with this dataset is 0.917 with threshold set to $\theta = 0.817100$. The values of precision and recall for this particular optimum are denoted as coordinates of the only point in the Figure 5.1. With the threshold set, we can investigate the absolute values for each class in the confusion matrix presented in table 5.1.

Table 5.1: Standard control dataset: confusion matrix

		True values	
		Negative	Positive
Predicted values	Negative	3,151,626	1,218
	Positive	209	2,947

The TP value of 2,947 is equal to the TP rate of 93.38 % which we consider a good result relative to the amount of data the model was trained on. The FP value of 1,218 indicates that if the model was used in a practical scenario, we would get on average less than one false-positive alert per positive sample, which is also a very satisfying number because, for every request, the operating staff would get at most two results for further inspection.

5.2   Dataset with Slowed Down Speed

To verify that our model is performing well also with higher latencies due to natural congestion in the network, we artificially slowed down the upload and download speed at our Tor exit relay from 1 Mb/s to 500 Kb/s. In this experiment, we captured 2,005 positive samples, from which we generate 2,005,000 samples setting $N_{neg} = 999$.

The best F_4 achieved is 0.903 with the corresponding threshold being not too far from the previous experiment, $\theta = 0.805722$. The Precision – Recall plot is presented in Figure 5.2. As denoted, the best recall is 0.925, which proves that delays in the network do not have any significant impact on the quality of the classifier's predictions. This is not surprising, since the data are preprocessed in such a fashion that they do not include any absolute timestamps; all timestamps were converted to time deltas between successive packets.
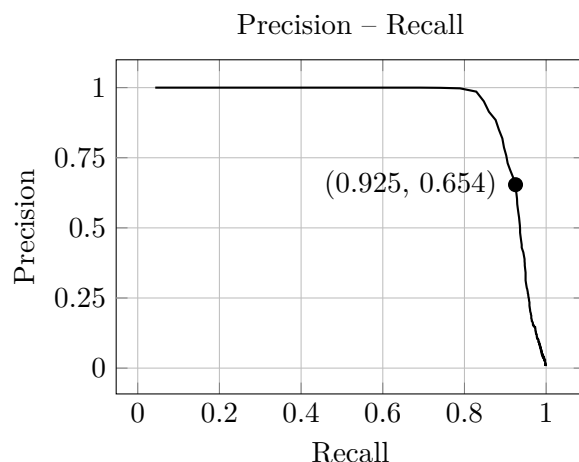
Precision – Recall

Figure 5.2: Slowed down control dataset: Precision – Recall

With the amount of False Positive samples being around half of the True Positive samples, this is still a reasonable price for less than 10 % of not captured positives.

5.3 External Dataset

To verify the universality of our model, we also used a dataset collected by our colleague for an entirely different project. This dataset, although smaller, contains a different type of user behavior as standard web browsing using the Tor network. The samples include watching videos, chatting through social networks, listening to music through the streaming platform, etc. If our classifier is able to distinguish correlation also in this dataset, it means that it is independent of the type of browsing and can be used for more general purposes.

The dataset contains 21 positive samples, thus generating the overall size of 441 samples with $N_{neg} = 20$. The best F_4 score has value 0.867 with best recall being 0.952 as depicted on Figure 5.3. In absolute values, the classifier successfully detected 20 out of 21 positive samples. It, however, produced 36 False Positive alerts. This indicates that the classifier was not trained on such type of data (which is true), but still can detect more than 95 % of correlated samples in them.

5.4 Negative Samples from the Same Domain

After consulting with the thesis advisors, we ran a small experiment on the top 10 Alexa's websites. We visited each of these websites 20 times in a row and set our preprocessing process to generate negative samples only from the
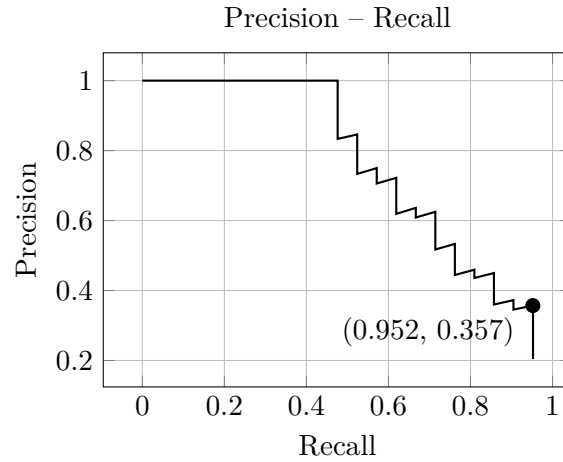
Figure 5.3: External control dataset: Precision – Recall

same website as a positive sample originated. This way, a negative sample's Tor exit part contains traffic with the same destination server, as a positive sample, only captured at a different time.

Although the resulting dataset is small (only 6,214 samples), the results of the experiment show that the model can differentiate well between connections to the same website, even seconds apart.

Table 5.2: Negative samples from the same domain: confusion matrix

		True values	
		Negative	Positive
Predicted values	Negative	5,975	0
	Positive	2	237

The recall has a value of 0.992, which corresponds to only 2 missed positive samples as shown in table 5.2. On the other hand, the precision is straight 1.0, meaning 0 False Positives. Bearing in mind the scale of the experiment, we can proclaim that the result of this experiment is promising and a greater-scale experiment should be made to verify this theory.

5.5 All Datasets Together

To summarize the efficiency of our classifier, we ran the last experiment, combining and mixing the samples from all previous datasets. These together

contain 9,823 positive captured samples, from which 2,946,900 were generated with $N_{neg} = 299$. We could not set N_{neg} higher due to the memory limit.

The best F_4 score achieved has a value of 0.944, which is a very satisfactory result. This value is derived from the recall of value 0.958 and the precision of value 0.770 as shown in Figure 5.4.
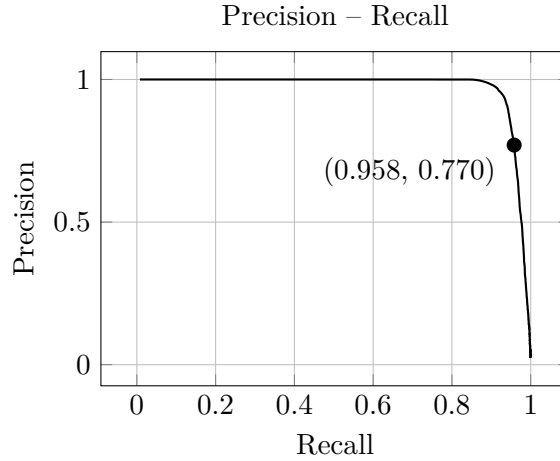

Figure 5.4: All datasets: Precision – Recall

Table 5.3: All samples: confusion matrix

		True values	
		Negative	Positive
Predicted values	Negative	2,934,261	2,816
	Positive	415	9,408

To break down the summary statistics into concrete numbers, we present the confusion matrix in table 5.3. Out of 9,823 positive samples, only 415 were not detected, which represent only 4 %, which is a very good result. The False Positive value of 2,816 represents around 23 % of all positive alerts. This means that around each 4th positive alert would need to be examined.

As we could not convert the original dataset used in the experiments by Nasr et al. [19] to our format, we, unfortunately, could not directly compare the performance of the two classifiers. We can, however, provide the reader with the ROC Curve, depicted in Figure 5.5, which shows the relation between False Positive Rate and True Positive Rate. Note that the False Positive Rate

axis is logarithmic. The thick solid line represents the ROC Curve of our classifier, whereas the dashed line represents the ROC of the original DeepCorr classifier using the flow length $l = 100$. We, however, do not conclude anything from this comparison, since both classifiers were used on different datasets. Converting our dataset to the format used by Nasr et al. for comparison is possible and on the recommendation of the supervisor will be done as future work.
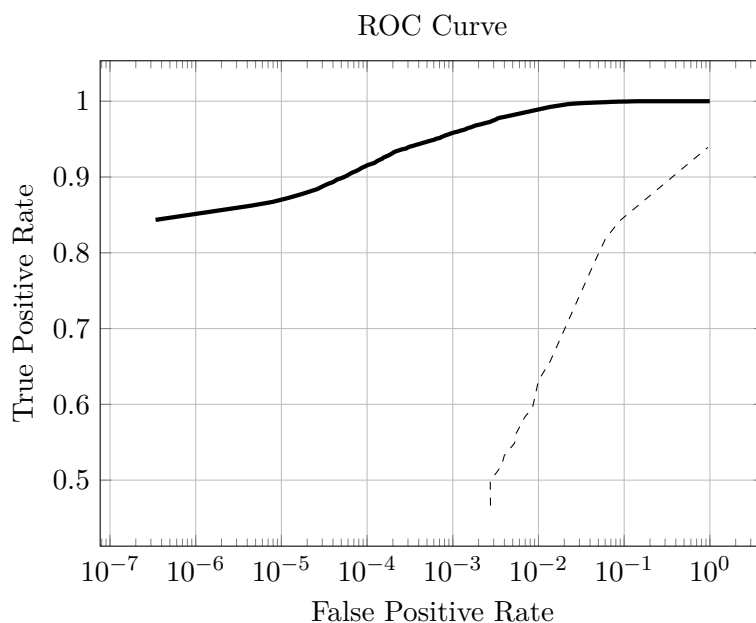

Figure 5.5: All datasets: ROC Curve

As of execution time, the whole classifying part of the algorithm took 440 seconds, meaning that classification of one sample took in average 0.149 ms to complete. This compared to RAPTOR with its 0.8 ms and original DeepCorr with 2 ms is a significant speed up [19]. A possible explanation might include the fact that our experiments were running at Google's Colab machines, equipped with high-speed GPU units.

5.6 Future Work

The outcomes of this thesis are mainly related to the experiments that evaluate a feasibility of the machine learning based detection of correlated connections. Additionally, thanks to a sophisticated system of network monitoring, and fast execution of the prediction phase of the neural network, the developed software prototype showcased in this work is near to be deployed in a real-world scenario. The real deployment, however, is out of scope of this thesis and is left as a future work. To achieve this, we will need to capture even more

various connections to accurately grasp the versatility of the Tor network and redesign the system to a shape, in which it can be deployed in a distributed way. There are still many further experiments we plan to execute. Most importantly, we need to compare the performance of our classifier with the one of Nasr et al. [19] to prove that we can achieve a comparable accuracy with only one-third of the length of flows used. Furthermore, we want to experiment with even shorter flows, shortening the input data to up to 30 packets, which can further improve the performance of the working classifier.

Conclusion

Privacy is the most important feature of the anonymization services and tools such as Tor, which is the main focus of this thesis. However, our experiments discovered a possibility to identify correlated parts of the user's connection that can disclose a target of the communication. The main goal of this work was to present the reader with a novel correlation attack on the Tor network based on the power of Deep Learning and extended IP flow data.

The theoretical part of this thesis described the principle of the Tor technology and general information on Convolutional Neural Networks feasible for Tor traffic analysis. We demonstrated that the correlation of Tor traffic does not need to be based only on raw network packets, as it is commonly used in the existing published papers, but it can benefit from using IP flow data, i.e., aggregated data, to train and test the model. Using such aggregated data significantly reduces the required hardware resources for practical deployment in high-speed networks.

The thesis also describes comprehensive experiments that evaluate the performance of the designed classification algorithm. The experiments were performed using newly created datasets from real network traffic captured on a deployed Tor relay. The experiments show promising results with over 90 % accuracy of correctly identified correlated parts of the client's connections. That means for every pair of connections observed from different observation points, it is most likely that the algorithm will correctly decide whether they belong to the same communication between the client and the server. The positive result discloses a particular target of a client, and thus represents a possible privacy weakness of the whole anonymization process.

We aim to spark the debate about how long can Tor, with its current design, maintain the anonymity of its users. If a research team is able to correlate with such accuracy, there is no doubt that state agencies can do the same, having the access to a few network bottlenecks. Therefore, as long as the Tor design does not change, the feasibility of these attacks will only grow, as the Deep Learning algorithms are advancing rapidly. From this point of

view, Tor users will need to either alter their behavior, which is very unlikely, or we will come to the point, where the design of the anonymization networks, such as Tor, will need to change inducing some kind of randomness, in order to defeat the correlation attacks once and for all.

Bibliography

[1] Inc., T. T. P. History. [online], [cit. 2021-04-14]. Available from: `https://www.torproject.org/about/history/`

[2] Dingledine, R.; Mathewson, N.; et al. Tor: The Second-Generation Onion Router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA: USENIX Association, Aug. 2004. Available from: `https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router`

[3] Wikipedia contributors. Tor (anonymity network) — Wikipedia, The Free Encyclopedia. 2021, [cit. 2021-04-21]. Available from: `https://en.wikipedia.org/w/index.php?title=Tor_(anonymity_network)&oldid=1019232296`

[4] Kastne, E. History of the dark web. [online], Feb 2020, [cit. 2021-04-14]. Available from: `https://www.soscanhelp.com/blog/history-of-the-dark-web`

[5] The Tor Project Inc. Tor Metrics: Servers. [online], [cit. 2021-04-14]. Available from: `https://metrics.torproject.org/networksize.html?start=2021-01-14&end=2021-04-14`

[6] Perry, M.; Clark, E.; et al. The Design and Implementation of the Tor Browser. [online], June 2018, [cit. 2021-04-21]. Available from: `https://2019.www.torproject.org/projects/torbrowser/design/`

[7] Evers, B.; Hols, J.; et al. Thirteen Years of Tor Attacks. 2016.

[8] Pries, R.; Yu, W.; et al. A New Replay Attack Against Anonymous Communication Networks. 06 2008, pp. 1578 – 1582, doi:10.1109/ICC.2008.305.

[9] Yang, M.; Luo, J.; et al. De-anonymizing and countermeasures in anonymous communication networks. *IEEE Communications Magazine*, volume 53, no. 4, 2015: pp. 60–66, doi:10.1109/MCOM.2015.7081076.

[10] Ling, Z.; Luo, J.; et al. A New Cell-Counting-Based Attack Against Tor. *IEEE/ACM Transactions on Networking*, volume 20, no. 4, 2012: pp. 1245–1261, doi:10.1109/TNET.2011.2178036.

[11] Loesing, K. Analysis of Circuit Queues in Tor. August 2009. Available from: `https://research.torproject.org/techreports/bufferstats-2009-08-25.pdf`

[12] Bauer, K.; McCoy, D.; et al. Low-Resource Routing Attacks against Tor. WPES '07, New York, NY, USA: Association for Computing Machinery, 2007, ISBN 9781595938831, p. 11–20, doi:10.1145/1314333.1314336. Available from: `https://doi.org/10.1145/1314333.1314336`

[13] Wang, X.; Luo, J.; et al. A Potential HTTP-Based Application-Level Attack against Tor. *Future Gener. Comput. Syst.*, volume 27, no. 1, Jan. 2011: p. 67–77, ISSN 0167-739X, doi:10.1016/j.future.2010.04.007. Available from: `https://doi.org/10.1016/j.future.2010.04.007`

[14] Arp, D.; Yamaguchi, F.; et al. Torben: A Practical Side-Channel Attack for Deanonymizing Tor Communication. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, New York, NY, USA: Association for Computing Machinery, 2015, ISBN 9781450332453, p. 597–602, doi: 10.1145/2714576.2714627. Available from: `https://doi.org/10.1145/2714576.2714627`

[15] Blond, S.; Manils, P.; et al. One Bad Apple Spoils the Bunch: Exploiting P2P Applications to Trace and Profile Tor Users. *4th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET '11)*, 03 2011.

[16] Troncoso, C.; Danezis, G. The Bayesian Traffic Analysis of Mix Networks. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, New York, NY, USA: Association for Computing Machinery, 2009, ISBN 9781605588940, p. 369–379, doi: 10.1145/1653662.1653707. Available from: `https://doi.org/10.1145/1653662.1653707`

[17] Murdoch, S.; Zielinski, P. Sampled Traffic Analysis by Internet-Exchange-Level Adversaries. 06 2007, ISBN 978-3-540-75550-0, pp. 167–183, doi: 10.1007/978-3-540-75551-7_11.

[18] Sun, Y.; Edmundson, A.; et al. RAPTOR: Routing Attacks on Privacy in Tor. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C.: USENIX Association, Aug. 2015, ISBN 978-1-939133-11-3, pp. 271–286. Available from: `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/sun`

[19] Nasr, M.; Bahramali, A.; et al. DeepCorr. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Jan 2018, doi:10.1145/3243734.3243824. Available from: `http://dx.doi.org/10.1145/3243734.3243824`

[20] McCullum, N. Deep Learning Neural Networks Explained in Plain English. [online], [cit. 2021-04-14]. Available from: `https://www.freecodecamp.org/news/deep-learning-neural-networks-explained-in-plain-english`

[21] Deshpande, A. A Beginner's Guide To Understanding Convolutional Neural Networks Part 2. [online], [cit. 2021-04-17]. Available from: `https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/`

[22] Brownlee, J. Crash Course On Multi-Layer Perceptron Neural Networks. *Machine Learning Mastery [online]*, May 2016, [cit. 2021-04-19]. Available from: `https://machinelearningmastery.com/neural-networks-crash-course/`

[23] Hornik, K.; Stinchcombe, M.; et al. *Multilayer feedforward networks are universal approximators*, volume 2. 1989, pp. 359–366.

[24] Srinivasan, A. V. Stochastic Gradient Descent — Clearly Explained !! *towards data science [online]*, Sep 2019, [cit. 2021-04-19]. Available from: `https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31`

[25] Saha, S. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. *towards data science [online]*, Dec 2018, [cit. 2021-04-16]. Available from: `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`

[26] Brownlee, J. How Do Convolutional Layers Work in Deep Learning Neural Networks? *Machine Learning Mastery [online]*, Apr 2019, [cit. 2021-04-17]. Available from: `https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/`

[27] Goodfellow, I.; Bengio, Y.; et al. *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[28] Pandey, S. How to choose the size of the convolution filter or Kernel size for CNN? *Analytics Vidhya [online]*, Jun 2020, [cit. 2021-04-17]. Available from: `https://medium.com/analytics-vidhya/how-to-choose-the-size-of-the-convolution-filter-or-kernel-size-for-cnn-86a55a1e2d15`

[29] Li, F.-F.; Krishna, R.; et al. CS231n Convolutional Neural Networks for Visual Recognition. [online], [cit. 2021-04-17]. Available from: `https://cs231n.github.io/convolutional-networks/`

[30] Hofstede, R.; Čeleda, P.; et al. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, volume 16, no. 4, 2014: pp. 2037–2064.

[31] Edwards, J. A Brief History of Network Monitoring. 2021, [cit. 2021-04-21]. Available from: `https://www.whatsupgold.com/blog/a-brief-history-of-network-monitoring`

[32] Mills, C.; Hirsh, D.; et al. RFC1272: Internet Accounting: Background. 1991.

[33] Leinen, S. Evaluation of candidate protocols for IP flow information export (IPFIX). Technical report, RFC 3955, October, 2004.

[34] Claise, B.; Trammell, B.; et al. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. 09 2013.

[35] Minarik, P. Continuous packet capture or flow monitoring? 2021, [cit. 2021-04-27]. Available from: `https://www.flowmon.com/en/blog/continuous-packet-capture-of-flow-monitoring`

[36] wireshark.org. *Tshark manual [online]*. [cit. 2021-03-28]. Available from: `https://www.wireshark.org/docs/man-pages/tshark.html`

[37] Selenium. *Selenium manual [online]*. [cit. 2021-03-28]. Available from: `https://github.com/SeleniumHQ/selenium`

[38] Cejka, T.; Bartos, V.; et al. NEMEA: A Framework for Network Traffic Analysis. In *12th International Conference on Network and Service Management (CNSM 2016)*, 2016, doi:10.1109/CNSM.2016.7818417. Available from: `http://dx.doi.org/10.1109/CNSM.2016.7818417`

[39] CESNET. *ipfixprobe manual [online]*. [cit. 2021-04-07]. Available from: `https://github.com/CESNET/ipfixprobe`

[40] Koech, K. E. Cross-Entropy Loss Function. *towards data science [online]*, Oct 2020, [cit. 2021-04-11]. Available from: `https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e`

[41] Kingma, D. P.; Ba, J. Adam: A Method for Stochastic Optimization. 2017, 1412.6980.

Acronyms

AS Autonomous System

BGP Border Gateway Protocol

CNN Convolutional Neural Network

FFNN Feed-forward Neural Network

FN False Negative

FP False Positive

MLP Multi-layer Perceptron

RP Rendezvous Point

OR Onion Router

OS Onion Service

SGD Stochastic Gradient Descent

SNMP Simple Network Management Protocol

TN True Negative

TP True Positive

VPN Virtual Private Network

VPS Virtual Private Server

Contents of Enclosed SD-Card

```
  readme.txt ...................... the file with CD contents description
├─ classifier
│  ├─ capture-and-preprocessing............... tools for dataset capture
│  └─ deep-learning ................ the deep learning model and classifier
├─ src ...................................... the directory of source codes
│  └─ thesis ............. the directory of LaTeX source codes of the thesis
└─ text ....................................... the thesis text directory
   ├─ thesis.pdf........................... the thesis text in PDF format
   └─ thesis.ps............................. the thesis text in PS format
```