FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

# ASSIGNMENT OF MASTER'S THESIS

| | |
|---|---|
| **Title:** | Detection of HTTPS brute-force attacks in high-speed computer networks |
| **Student:** | Bc. Jan Luxemburk |
| **Supervisor:** | Ing. Karel Hynek |
| **Study Programme:** | Informatics |
| **Study Branch:** | Computer Security |
| **Department:** | Department of Information Security |
| **Validity:** | Until the end of summer semester 2020/21 |

## Instructions

Study the area of network monitoring based on deep packet inspection and (extended) IP Flows.
Study the HTTP(S) protocol and a principle of brute-force attacks over HTTPS protocol.
Set up a virtual environment with existing tools for penetration testing (such as Hydra) to create a dataset of brute-force attacks against selected virtual servers (e.g., Wordpress, Joomla).
Analyze the created datasets and focus on significant characteristics of brute-force attacks.
Design an algorithm of an automatic attack detection of brute-force attacks (inspired by [1]) based on observed network traffic.
Develop a software prototype capable of processing of real network traffic.
Evaluate the prototype and its precision with the data provided by the supervisor of this thesis.

## References

[1] Hofstede, Rick & Jonker, Mattijs & Sperotto, Anna & Pras, Aiko. (2017). Flow-Based Web Application Brute-Force Attack and Compromise Detection. Journal of Network and Systems Management. 10.1007/s10922-017-9421-4.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 5, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

# Detection of HTTPS brute-force attacks in high-speed computer networks

***Bc. Jan Luxemburk***

Department of Information Security
Supervisor: Ing. Karel Hynek

May 31, 2020

# Acknowledgements

I would like to thank Ing. Karel Hynek for his guidance and helpful approach.

# Declaration

**Citation of this thesis**

Luxemburk, Jan. *Detection of HTTPS brute-force attacks in high-speed computer networks*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

# Abstract

This thesis presents a review of flow-based network threat detection, with the focus on brute-force attacks against popular web applications, such as WordPress and Joomla. A new dataset was created that consists of benign backbone network traffic and brute-force attacks generated with open-source attack tools. The thesis proposes a method for brute-force attack detection that is based on packet-level characteristics and uses modern machine-learning models. Also, it works with encrypted HTTPS traffic, even without decrypting the payload. More and more network traffic is being encrypted, and it is crucial to update our intrusion detection methods to maintain at least some level of network visibility.

*Keywords* network monitoring, brute-force attacks, HTTPS, IPFIX

# Abstrakt

Tato práce představuje přehled metod pro detekci síťových hrozeb se zaměřením na útoky hrubou silou proti webovým aplikacím, jako jsou WordPress a Joomla. Byl vytvořen nový dataset, který se skládá z provozu zachyceného na páteřní síti a útoků generovaných pomocí open-source nástrojů. Práce přináší novou metodu pro detekci útoku hrubou silou, která je založena na charakteristikách jednotlivých paketů a používá moderní metody strojového učení. Metoda funguje s šifrovanou HTTPS komunikací, a to bez nutnosti dešifrování jednotlivých paketů. Stále více webových aplikací používá HTTPS pro zabezpečení komunikace, a proto je nezbytné aktualizovat detekční metody, aby byla zachována základní viditelnost do síťového provozu.

*Klíčová slova* monitorování počítačových sítí, útoky hrubou silou, HTTPS, IPFIX

# Contents

# List of Figures

# List of Tables

# Introduction

The state of security of Internet-connected systems is constantly changing. As technology advances, and as more and more devices are connected to the Internet, the target attack surface is rapidly increasing, and the motivation of malicious attackers is only getting bigger. Imagine hacking a car or a medical device, the potential caused harm is obviously enormous. Organizations have to adapt, but the sad reality is that defenders will always be playing catch-up. This led to a new threat-centric approach, which embraces that defense cannot be solely based on prevention. No matter how strong the defenses are or what proactive steps have been taken, a motivated attacker will eventually find a way to get in. Prevention eventually fails. That is why detection, response, and containment are necessary for the incidents that will occur, and that is why the practice of network security monitoring is a cornerstone of security for every organization.

Between 2015 and 2019, the percentage of encrypted web traffic from the Firefox web browser had raised from 35% to 85% [1], and it is expected to continue growing. This is good news, but it brings a new challenge to network operators. There is less visibility into the network, and therefore it is harder to distinguish between legitimate and malicious traffic. Moreover, in 2018, a new version of the TLS protocol was released, which even more reduced the available information about encrypted web traffic. It is hence the focus of current research to find novel ways of effective network monitoring.

This thesis focuses on one particular attack—the brute-force attack against web applications. A brute-force attack is a method of trial and error. The attackers run automated scripts that try thousands of passwords each second in order to take over the target accounts. Hacked accounts are then used for a range of illegal activities such as spamming, malware hosting, botnet participation, or financial fraud. With the majority of web traffic being encrypted, it is even more difficult to detect such attacks on the network level. The thesis proposes a new detection method that is able to recognize brute-force attacks inside encrypted HTTPS communication.

Chapter 1

# Goals and Approach

## 1.1 Problem Description

This thesis is focused on brute-force attack detection in encrypted HTTPS communication. Encryption provides confidentiality—without the knowledge of the secret key, it is impossible to reveal the plaintext. But does that mean we cannot predict whether an encrypted network session contains regular web browsing or an active brute-force attack? This work will explore whether such detection is achievable.

## 1.2 Objective of Thesis

The objective of this work is to examine the possibilities of HTTPS web traffic monitoring and to design and implement a detection mechanism for brute-force attacks against web applications, in particular against popular content management systems, such as Wordpress, Joomla, and Moodle.

The goals of this thesis can be summarized as follows:

1. Provide an analysis of the architecture of flow-based network security monitoring.
2. Review the available open-source brute-force tools, discuss their capabilities, and compare them.
3. Create an environment for generating brute-force traffic datasets.
4. Identify flow features suitable for the classification task between normal (benign) and brute-force traffic. Define the pre-processing steps and evaluate different machine-learning models.
5. Design and implement a software prototype of the brute-force detection method.

## 1.3 Outline of Thesis

This thesis is divided into five chapters. Chapter 2 gives the theoretical background about the related topics: flow-based network monitoring, extended flow features, threat detection, and TLS and HTTPS protocols. The dataset structure and the process of brute-force traffic generation are presented in Chapter 3, together with the design of the detection method. The method's design is divided into several components: flow aggregation, selected extended flow features, and used machine-learning models. Chapter 4 presents a production architecture and an implementation of the proposed method as a detector module into the NEMEA system. Chapter 5 concludes the thesis with the evaluation of different machine-learning models and discussion of future work.

Chapter 2

# Background

This Chapter summarizes the theoretical background important for the area of flow-based threat detection, with the focus on flow monitoring architecture in Section 2.1.1, on extended flow features in Section 2.1.3, on flow-based threat detection in Section 2.1.4, on TLS and HTTPS protocols in Section 2.2, and finally on web application brute-force attacks in Section 2.3.

## 2.1 Network Security Monitoring

Security monitoring is a crucial part of network defense. Organizations should accept that their networks will be eventually compromised and put more focus and resources on detection and response mechanisms. In [2], network security monitoring is defined as a cyclic process that consists of three phases: collection, detection, and analysis. The collection phase is about defining threats, assessing risks, and selecting appropriate data to be collected. It is not advised to collect all data sources available, but rather be selective with specific threats in mind. There is a number of different kinds of data that can be collected: full packet capture, flow data, and log data. Full packet capture data, mostly know in the PCAP format, provide a complete transcript of communication between two network devices. Its high degree of detail makes it valuable for analysis, but otherwise, it is impractical for its huge size. Flow data represents a summary of communication between two endpoints. It usually includes information about the communicating hosts, their IP addresses and ports, the used protocol, timestamps, and the amount of transferred data. The detection is the process by which the collected data is examined and the way how alerts are generated based on observed events and a set of signature rules. The generated alerts are then presented to security analysts for manual inspection or used for further automatic processing. In this work, I shall focus on network intrusion detection systems (NIDS) that process flow data in order to identify threats in monitored traffic.

### 2.1.1 Flow Monitoring Architecture

Flow monitoring has become the prevalent method for traffic monitoring in high-speed networks. While flow data does not provide the same level of detail as full packet capture, it still carries a lot of valuable information, and it is a much more scalable solution. The small size of flow records allows for large scale storage solutions with data retention in months or years, something that would impossible with full packet capture. This is frequently used to comply with data retention laws. For example, communication providers in Europe are required to retain connection data for a period of up to two years for the purpose of investigation of serious crime [4]. Flow monitoring is usually less privacy-sensitive because only packet headers are processed. However, this might not be that case much longer because more application information (e.g., HTTP user agents, server hostnames) is included in the flow data. The general scheme of the monitoring process is shown in Figure 2.1 and Figure 2.2.

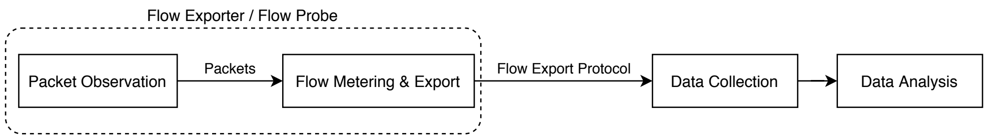


Figure 2.1: Flow monitoring stages.

A typical flow monitoring architecture is defined in *Flow Monitoring Explained* [4], and it consists of four stages: Packet Observation, Flow Metering & Export, Data Collection, and Data Analysis.

#### Packet Observation

Packet Observation is the process of capturing packets from an observation point, which can be *"line to which a probe is attached; a shared medium, such as an Ethernet-based LAN; a single port of a router; or a set of interfaces (physical or logical) of a router"* [3]. Accurate packet timestamps are essential for further processing, for example, when packets from different sources are later merged. Packet capture devices can be positioned in-line or in mirroring mode. An in-line network tap is a hardware device that is directly connected to the monitored link. It duplicates all traffic passing through and provides an additional port for the capture device. Alternatively, many packet forwarding devices can be configured to mirror all traffic to a specific port to which a capture device is connected. This method is called port mirroring or SPAN, and while it is easier to configure, it may introduce delay, jitter, or miss some packets altogether [5], which is something highly undesirable in high-security environments.

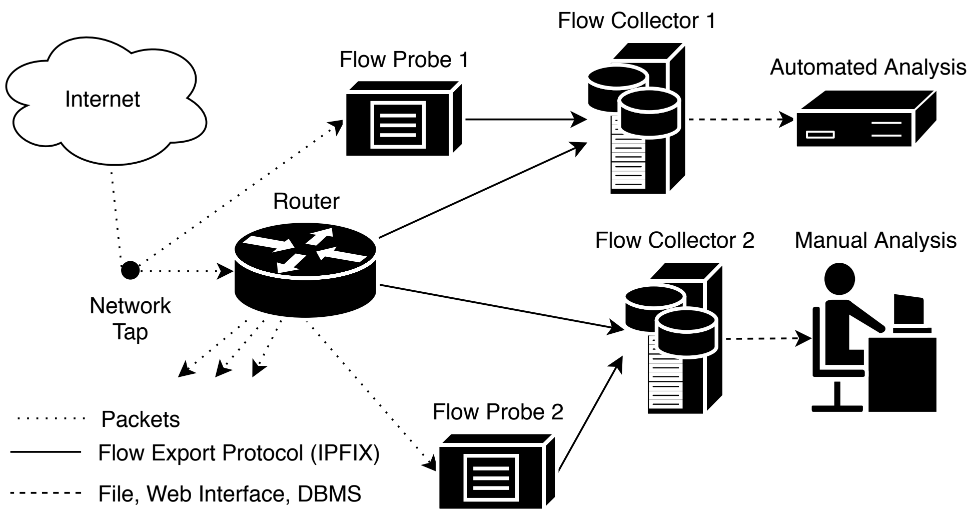


Figure 2.2: Various flow monitoring setups [4].

### Flow Metering & Export

In the Flow Metering & Exporting stage, packets are first aggregated into flows. A flow is defined in [3] as *"a set of IP packets passing an observation point in the network during a certain time interval, such that all packets belonging to a particular flow have a set of common properties"*. These common properties are called flow key. A traditional "5-tuple" flow key is: source and destination IP addresses, source and destination ports, and a transport protocol. The time interval defining a flow generally spans from the first observed packet of the flow to one of three events: either the natural end of the flow, the idle timeout of the flow, or the active timeout of the flow [6]. The natural end of a flow is determined by observing the state of connection-oriented protocols. In the case of TCP, a flow will end when a FIN termination handshake or a RST packet is seen. The idle timeout is the longest period of time between packets, after which the flow will be considered idle and will end. This is the natural way to expire flows in connection-less protocols such as UDP. The active timeout is the longest lifetime a flow is allowed to have.

An entry per flow is stored in a flow cache table, and once a flow is determined to be complete, it is exported via a flow exporting protocol. The exported data include both characteristic properties of the flow (e.g., IPs, port numbers) and measured properties (e.g., packet and byte counters). A dedicated hardware device for packet capture and flow export is called flow probe. Flows used to be unidirectional and used to represent only one direction of the connection. Lately, however, bidirectional flows are preferred because *"many flow analysis tasks benefit from the association of the upstream and downstream flows of bidirectional communication, e.g., when separating answered and unanswered TCP requests or calculating round trip times"* [7].

Asymmetric routing is an issue, as only one of the directions may be available at the observation point. In that case, it is necessary to merge unidirectional flows (uniflows) into bidirectional flows (biflows) at a flow collector.

#### Data Collection

The next stage in the flow monitoring architecture is Data Collection. Flow collectors receive, store, and process data from one or more flow exporters. Common tasks performed at the flow collectors are data compression, aggregation [8], summary generation, and data anonymization [9]. It is crucial for flow collectors to support the same export protocol features as the used flow exporters, such as data encoding, transport protocol, and all exported fields.

#### Data Analysis

In the final Data Analysis stage, flow data is either automatically processed or inspected manually. There are three main application areas for flow data analysis: summaries & reporting, threat detection, and performance monitoring. The focus of this thesis is threat detection, which is discussed more in Section 2.1.4.

### 2.1.2 NetFlow & IPFIX

The two main flow exporting protocols are NetFlow from Cisco and the IP Flow Information Export (IPFIX) protocol standardized by the Internet Engineering Task Force (IETF). NetFlow v5 was introduced in 2002 and was widely implemented in many packet forwarding devices. NetFlow v5 was later obsoleted by the more flexible NetFlow v9, which added support for adaptable data formats through templates, as well as support for IPv6, VLANs, and MPLS. NetFlow v9 was selected in 2004 as the basis for the IPFIX protocol, which was finished in RFC 7011 in 2013 [3].

#### IP Flow Information Export Protocol

IPFIX is an *"unidirectional, transport-independent protocol with flexible data representation"* [10]. IPFIX is concerned only with the transport of flow data, leaving the flow measurement as an implementation detail. The basic unit of data transfer is a message. A message contains a header and one or more sets, which contain records. The message header contains the protocol version, message length, export time, sequence number, and an observation domain ID. A set may be either (1) a template set, containing templates; (2) a data set, containing data records (i.e., flow records); or (3) an options template set with metadata information. Each set has an ID in its header that identifies the set type. Set IDs 2 and 3 are used for template sets and options template sets. Set IDs 256–65535 are used for data sets, and the ID refers to the template, which describes the records in that data set. A template is essentially an ordered list of information elements identified by a template ID. The relation between data sets and templates is shown in Figure 2.3.

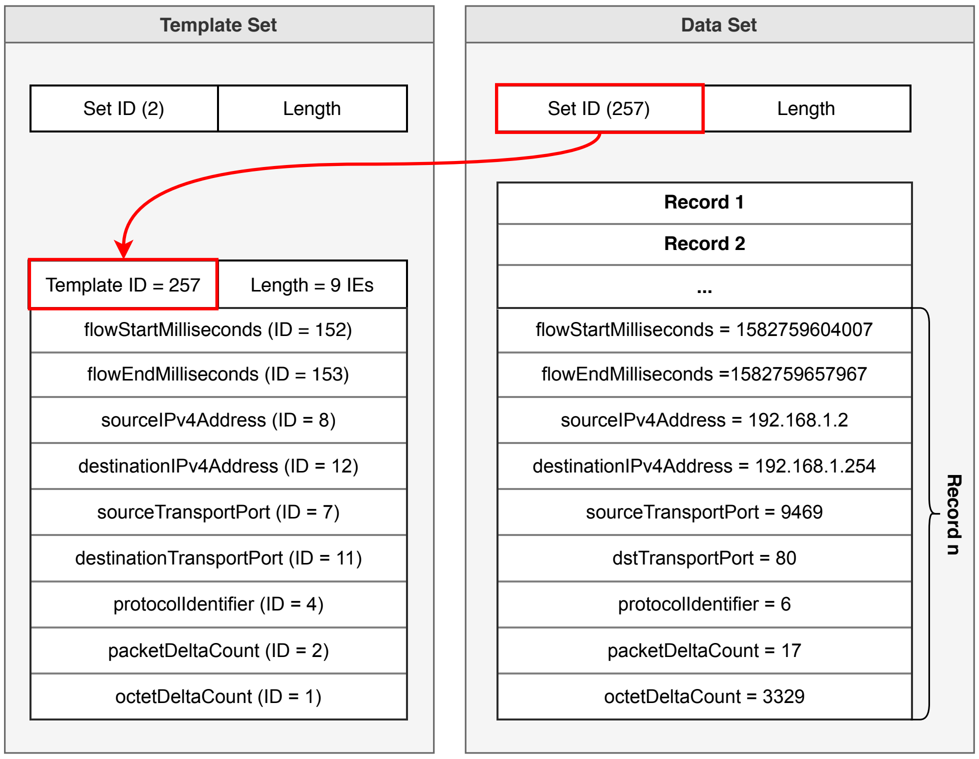

Template Set
Set ID (2)
Length
Template ID = 257
Length = 9 IEs
flowStartMilliseconds (ID = 152)
flowEndMilliseconds (ID = 153)
sourceIPv4Address (ID = 8)
destinationIPv4Address (ID = 12)
sourceTransportPort (ID = 7)
destinationTransportPort (ID = 11)
protocolIdentifier (ID = 4)
packetDeltaCount (ID = 2)
octetDeltaCount (ID = 1)
Data Set
Set ID (257)
Length
Record 1
Record 2
...
flowStartMilliseconds = 1582759604007
flowEndMilliseconds =1582759657967
sourceIPv4Address = 192.168.1.2
destinationIPv4Address = 192.168.1.254
sourceTransportPort = 9469
dstTransportPort = 80
protocolIdentifier = 6
packetDeltaCount = 17
octetDeltaCount = 3329
Record n

Figure 2.3: The relation between an IPFIX data set and a template.

### Information Elements

Fields that can be exported in IPFIX flow records are named information elements. An information element (often abbreviated as IE) represents a named data field with a specific data type and meaning. IPFIX provides a registry of standard IEs administered by the Internet Assigned Numbers Authority (IANA) [11]. A subset of IEs is shown in Table 2.1, which is often considered the smallest set of IEs for describing a flow. Besides IANA IEs, enterprise-specific IEs can be defined, allowing for new fields to be specified without any alternations to IANA's registry. IEs have a name, numeric ID, description, type, length (fixed or variable), status (current or deprecated), and an enterprise ID in case of enterprise-specific IEs [12]. Templates are built from IEs the following way. *"Each template has an IE count N followed by N IE specifiers, each made up of an IE number and length in bytes. The corresponding data record is then made up of the same order of fields of the specified lengths"* [10]. In RFC 6313 [13], an IPFIX extension was specified to support hierarchical structured data and variable-length lists. Three new IEs were defined: basicList, subTemplateList, and subTemplateMultiList. The basicList represents a list of zero or more instances of any IE and is primarily used for single-valued data types. Examples are a list of port numbers, a list of MPLS labels, a list of packet sizes. The subTemplateList represents a list of structured data defined by a template. SubTemplateMultiList is similar, but every element has a different template.

Table 2.1: Common IPFIX Information Elements.

| **ID** | **Name** | **Description** |
|---|---|---|
| 152 | flowStartMilliseconds | Timestamp of the flow's first packet |
| 153 | flowEndMilliseconds | Timestamp of the flow's last packet |
| 8 | sourceIPv4Address | Source address in the packet header |
| 12 | destinationIPv4Address | Destination address in the packet header |
| 7 | sourceTransportPort | Source port in the transport header |
| 11 | destinationTransportPort | Destination port in the transport header |
| 4 | protocolIdentifier | IP protocol number in the packet header |
| 2 | packetDeltaCount | Number of packets for the flow |
| 1 | octetDeltaCount | Number of octets for the flow |

The IANA registry currently includes 491 information elements of diverse purposes [11]. Most of the IEs are defined by RFCs related to IPFIX standard, but also individuals and companies can propose new elements. As of March 2020, Cisco authored most of the non-RFC elements in the registry. The majority of IEs in the registry are related to network protocols, such as IP, TCP, VLAN, MPLS, BGP, NAT, ICMP. There are elements concerned with IPFIX exporting process itself. An example is a flow probe IP address, its network interface, up-time, and the number of dropped packets. Another category of IEs is application layer elements. These are not yet very common in the registry, and exporters have to define them as enterprise-specific elements. For example, an open-source flow exporter YAF supports, among others, these application protocols: FTP, HTTP, IMAP, SIP, SMTP, DNS, TLS [14]. For each protocol, it extracts relevant information from the packets, such as HTTP headers, DNS queries, or TLS certificate, server name, client ciphers. This approach is called deep packet inspection (DPI). Although seemingly opposing, DPI and flow monitoring are increasingly united for increased visibility in networks.

### 2.1.3 Extended Flow Features

Researchers are focused on defining new network traffic features that could be used for detection and classification tasks. An underlying assumption is that traffic at the network layer has statistical properties that are unique for certain classes of applications and that enable different source applications (or different actions in applications) to be distinguished from each other. Almost 250 potential features for classification of flows are listed in [15]. These include simple statistics (mean, standard deviation, minimum, maximum, quartiles) about packet-level characteristics, and also information derived from the transport protocol, such as the number of retransmissions or the number of roundtrips. More characteristics can be derived from the nature of data exchange. A lifetime of a flow can be divided into three modes, and features can be related to the percent of the time a flow spends in them:

*idle*: no packets between client and server for more than a couple of seconds,

*interactive*: data packets moving in both directions, and

*bulk*: data packets in one direction and only acknowledgments in the other.

### Packet-Level Characteristics

Different applications show distinctive properties when their network traffic is analyzed in terms of packet sizes (PS) and inter-packet times (IPT), also called inter-arrival times (IAT). Another name used by researchers is the sequence of packet lengths and times (SPTL). A set of features is extracted from the first $N$ packets of the flow, considering only packets with a non-zero payload. Empty packets are mostly used to transmit connection state information, e.g., to acknowledge received data or to keep alive a session. They are related to the transport layer internals, as opposed to being linked to the way each specific application operates. TCP retransmissions are usually ignored. Formally these features can be defined as follows:

- $S = (s_1, s_2, \dots s_N)$, where $s_i$ is the payload size of the $i$-th packet.
- $T = (t_1, t_2, \dots t_N)$, where $t_i$ is the time between the $(i-1)$-th and the $i$-th packet, with $t_1$ zero.
- $D = (d_1, d_2, \dots d_N)$, where $d_i$ is the direction of $i$-th packet (upstream or downstream).

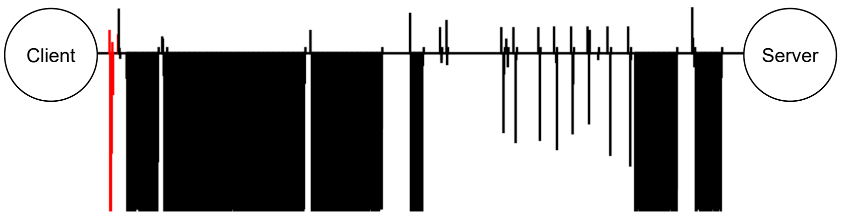
In [16], authors conclude that packet sizes usually carry more information about the source application. The reason is that inter-arrival times are affected by various factors, some that are useful for the classification, others are only noise without any value. The factors contributing to inter-arrival times are:

- the location of the communicating hosts with reference to the monitoring node: the response time depends on the route of the packets and on the number of intermediate nodes they pass through,
- the traffic condition of the network,

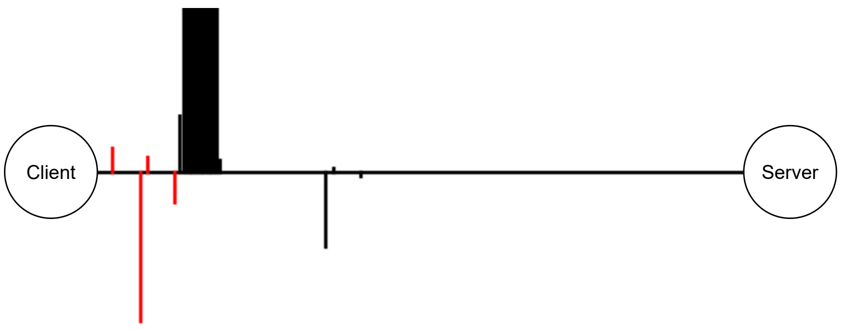
- the transmission direction of the $i$-th and $(i-1)$-th packets: when both the packets travel in the same direction, they are often the result of TCP segmentation, and they are sent sequentially with a small or any delay,
- the time required for the processing of the received data and for generating the response to send in the next packet, e.g., for reading or modifying a database, for computing cryptographic operations in an authentication phase, or for interacting with the user.

The first two factors do not give any valuable information about the underlying application or user actions. Still, it might be impossible to get rid of them, and the resulting IPT values will always contain noise.

The advantages of studying network traffic by observing IPT and PS values are the avoidance of any assumptions regarding the application layer protocols, and the possibility to study different kinds of traffic sources in the same manner [17]. There are multiple ways of how the packet-level characteristics can be used for statistical models and classification tasks. In [18], a Markov chain representation is used to model the SPLT data. *"For both the lengths and times, the values are discretized into equally sized bins, e.g., for the length data, 150-byte bins are used where any packet size in the range [0,150) will go into the first bin, any packet size in the range [150,300) will go into the second bin, etc. A matrix A is then constructed where each entry A[i,j], counts the number of transitions between the i'th and j'th bin. Finally, the rows of A are normalized to ensure a proper Markov chain".* The approach in [19] considers the joint distribution of PS and IPT. They apply strong discretization (binning) to the estimate of the joint probability density function and employ machine-learning algorithms like K-Nearest Neighbor(k-NN) and Support Vector Machines (SVM). In [19], a single source of traffic is modeled as a Hidden Markov Model (HMM). The traffic generated by a specific application is viewed as a sequence of (IPT, PS) pairs generated according to different distributions depending on the hidden state of the source. One HMM model for every application is trained in a learning phase. For the predictions, each HMM is used to compute the likelihood that the tested sequence belongs to the traffic typology associated with that HMM. The maximum likelihood then selects the best estimate for the traffic typology. An example of how packet-level characteristics can be used to build behavioral profiles of applications is shown in the next Figure 2.4.



(a) Google Search



(b) Bestafera Trojan Malware

Figure 2.4: TLS record lengths and inter-arrival times for a typical Google search and malicious data exfiltration from a trojan malware [20]. Two different TLS sessions are shown: a Google search in Figure 2.4a and a trojan-initiated connection in 2.4b. The x-axis represents time, the upward lines represent the size of packets that are sent from the client to the server, and the downward lines represent packets in the opposite direction. The red lines show unencrypted TLS handshake messages. The rest of the communication (black) is encrypted. The differences between those communication sessions can be explained as follows: *"the Google search follows a typical pattern: the client's initial request is in a small outbound packet, followed by large response spanning many MTU-sized packets. The several alternating packets are due to Google attempting to auto-complete a search while the user was still typing. The server that the trojan communicated with began by sending a packet containing a self-signed certificate, which can be seen as the first downward, thin red line in Figure 2.4b. After the handshake, the client immediately begins exfiltrating data to the server. Then after a pause, the server sent a regularly scheduled command and control message."* [20]

### Byte Distributions

The nature of the traffic within a flow can be deduced from the byte distribution of packet payloads and its Shannon entropy value. A common approach would be scanning through the first $n$ bytes of captured payload byte-by-byte and creating a histogram distribution within a 256-entry array. If the calculated entropy is then scaled to the range 0–255, a value of 255 would indicate a perfectly random set of data, while a value close to 0 would indicate an extremely redundant set of data with almost no information content. High entropy values (approximately above 230) indicate data that is either compressed or encrypted. Lower values centered around 140 likely indicate something such as an ASCII-based protocol, or English text [14].

### TLS Handshake Metadata

A TLS connection starts with a handshake between the client and server, which is then used to select the best mutually acceptable cryptographic ciphers, authentication methods, hashing algorithms, etc. This is conducted in the clear because the method of cryptography to use has yet to be determined. The TLS handshake will always begin with a Client Hello packet, which announces to the server the capabilities of the client, presented in preference order. By capturing those elements of the Client Hello packet, which remain static from session to session for each client application, it is possible to build a fingerprint to recognize a particular application on subsequent sessions. In a popular TLS fingerprinting tool JA3 [21], these fields are captured: TLS Version, Accepted Ciphers, List of Extensions, Elliptic Curves, and Elliptic Curve Formats. Fields are hashed with MD5 to generate a unique fingerprint. Fingerprints can then be annotated and shared with the security community on dedicated websites [22].

The most obvious use of TLS Fingerprinting is for passive detection. It enables the detection of a wide range of potentially unwanted traffic without requiring access to the hosts. The ability to detect malware or software such as Tor or PowerShell (which is sometimes used for malicious activities [23]) can be very valuable. Other potentially unwanted software can also be detected using this technique. *"The detection of software, which may not be malicious, but is out of context, could also be worthy of investigation and is simple to detect. For example, many interfaces should only be accessed by a particular client or set of clients. If a web server is expecting human interaction via a browser, detecting the fingerprint of wget could be significant; alternately, an Exchange server may only ever be accessed by Outlook, thus a connection from a Python script would be significant"* [24].

### 2.1.4 Threat Detection

Given the flow export devices are usually deployed at central observation points where traffic from a large number of hosts can be observer, the resulting flow data provides a comprehensive set of connection summaries that include information about which host has communicated with which other host, the number of packets and bytes involved, the number of connections, flags, top-talkers, etc. This kind of information is essential for forensics, network administration, and incident handling. Moreover, the strategical location of flow exporters makes them especially useful for the detection of DDoS attacks, network scans, worm spreading, and botnet communication. These attacks often result in many small flows with few packets because every connection uses a different source port number and creates a new flow. Such differences are used to distinguish between malicious and benign traffic. The rest of this section presents how different types of attacks can be detected based on flow characteristics.

#### DDoS

Four flow metrics that change significantly during a DDoS flooding attack have been identified in [25]: flow record creations per second, average flow duration, the average number of bytes per flow, and the average number of packets per flow. All but the average flow duration can be monitored on a flow exporter by using only counters. Based on these metrics, different attacks can be described in terms of traffic patterns. SYN flooding attacks result in *"a large flow count, yet small packet counts, as well as small flow and packet sizes and no constraints on the bandwidth and the total amount of packets. That pattern is significantly different from the one generated by an ICMP or UDP flooding attack, in which we have large bandwidth consumption and the transfer of a large number of packets"* [26]. A lightweight (i.e., with a minimal performance footprint) method for detecting large flooding attacks was proposed in [27]. The method measures the number of flow record creations and uses an anomaly-based approach called time-series forecasting, which uses previous measurements to forecast the next value. If the measured value does not lie within a certain range of the forecasted value, the sample is considered anomalous, and a potential flooding attempt has been detected. A firewall is notified of each anomaly, and the subsequent connections from the host that triggered the anomaly are blocked. This module can be directly integrated into a flow exporter, allowing for timely detection and timely mitigation.

#### Network Scans

Network scans are characterized by many small packets that probe the target systems. It is easy to imagine that network scanning can quickly create a large number of flows this way. Scans can be divided based on how they are targeted: (1) a host scanning a specific port on many destination hosts (horizontal scan), (2) a host scanning several ports on a single destination host (vertical scan), (3) a combination of both (block scan). Regardless of the type, network scanning will cause a variation in the flow traffic, but at the same time, scanning is less likely to have an impact on the total traffic volume in the network.

A computer worm is a malware program that replicates across networks to infect other computers. It explores the network in order to find more vulnerable systems. This target discovery process is similar to network scanning, and the detection can be the same. A totally different approach is based on a graph view of computer networks consisting of nodes (computers) and edges (time-series of communications between computers). A worm spreading through a network is detected by measuring a deviation from a baseline activity model of the network. *"Communications between computers which have not communicated in the past (new edges) can provide a strong statistical signal for detecting attackers. As they move laterally through a network, attackers tend to violate the historic connectivity patterns in the network. Knowledge of these patterns is perhaps the primary advantage defenders have over attackers"* [28]. The challenge is to handle updating of the parameters of baseline models in a way that balances acceptable false alarm rates with rapid updating to adjust for changes in the baseline behavior [28].

### Botnets

Botnets consist of malware-infected hosts that are controlled remotely using a command & control server (C&C). They have become one of the major security threats credited for DDoS attacks, spamming, phishing, and many other cybercrimes. Botnets rely on communication channels varying from centralized IRC or HTTP to decentralized peer-to-peer networks. Domain generation algorithms are commonly used by C&C servers to avoid hard-coded domain names and IP addresses in the malware executables because these would give defenders an easy way to blacklist and block the botnet communication. Detection of a botnet is relatively more difficult than the detection of network scanning and worms. Since bots are not longer harmful once the control server is isolated, a straightforward mitigation approach is to identify the control server, and either filter the traffic towards this server or take the server down with the help of law enforcement. C&C communication is usually different from communication with a regular server, and these differences can be seen on the flow level.

A large-scale botnet detection system based on flow data [29] is using three groups of features that allow distinguishing between C&C and benign communication: (1) Flow sizes in bytes; C&C communication flows are expected to be smaller to minimize their observable impact on the network, and the sizes should not fluctuate significantly. Furthermore, the regularity of flow size behavior over time can be measured by mean and standard deviation values, and repeating patterns that are common in C&C communication can be identified with an autocorrelation function over a sequence of flow sizes. (2) Client access patterns; all clients of a C&C server (i.e., bots) should exhibit similar access patterns, and they should connect to the control server in a periodic fashion. This can be identified by flow inter-arrival times, which is a sequence of time differences between consecutive flows. (3) Temporal behavior; benign traffic follows the well-known diurnal pattern—high volume during the day, and very little during the night. On the other hand, the majority of botnets is configured to contact the C&C independently of daytime, and the flows are therefore distributed uniformly throughout the day [29].

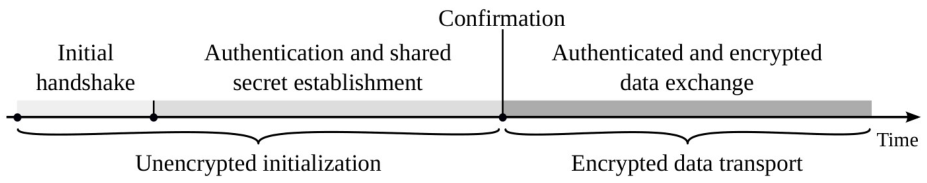

Confirmation
Initial handshake
Authentication and shared secret establishment
Authenticated and encrypted data exchange
Time
Unencrypted initialization
Encrypted data transport

Figure 2.5: A general scheme of network encryption protocols [30].

### Encrypted Traffic Analysis

Identifying threats in encrypted traffic is a significant challenge. Nevertheless, the security community has put forth several solutions to this problem. A solution standard in enterprise networks is called TLS inspection. It is a security process that allows enterprises to decrypt traffic, inspect the decrypted content for threats, and then re-encrypt the traffic before it enters or leaves the network. Once the traffic is decrypted, traditional signature-based methods, such as Snort [31] or Suricata [32], can be used for threat detection. Although this approach breaks the privacy of users, monitoring, detection, and mitigation of malware and forbidden user actions are legitimate needs (for example, when tracking illegal activities in the financial sector). Another method of identifying threats in encrypted traffic leverages flow data and encryption protocol metadata. Most of the encryption protocols can be divided into two main phases: the initialization of the connection and the transport of encrypted data, as depicted in Figure 2.5. The first phase can be further divided into algorithms negotiation, authentication of the communication parties, and establishment of shared secret keys, which are then used for encrypting and authenticating transferred data in the second phase.

The initialization phase is usually not encrypted, and a great deal of metadata can be extracted that identify different operating systems, web browsers, and other applications together with their versions. Researchers at Cisco [18] were able to identify 18 malware families in encrypted traffic. They used flow statistics, the sequence of packet lengths and times, byte distribution, and TLS metadata, such as the length of the public key, the selected cipher suite, the number of certificates, the number of subjectAltName names, certificate validity in days, and whether any certificate was self-signed [33]. Passive monitoring of certificates in the network enables not only the identification of communicating peers but also gives the ability to check whether certificates are valid and use proper security algorithms to fulfill local security policies.

Packets exchanged during the transport phase usually contain only the encrypted payload and information about the packet itself, such as the length and type. The packet lengths, however, still leak information about the nature of the traffic. Fingerprinting of HTTPS websites presented in [34] is based on the sequence of packet lengths. An attacker prepares a model of a target website in advance, which consists of packet size patterns for each page within the target website. By monitoring the sizes of packets coming from a victim, it is possible to identify a specific page that the victim is visiting even though the traffic is encrypted. Such an attack can be used for exposing personal details, including medical conditions, financial and legal affairs, or sexual orientation. Authors of the paper test this technique on *"three websites related to healthcare, since the page views of these websites have the potential to reveal whether a pending procedure is an appendectomy or an abortion, or whether a chronic medication is for diabetes or HIV/AIDS. We also examine legal websites, offering services spanning divorce, bankruptcy, and wills and legal information regarding LGBT rights, human reproduction, and immigration"* [34]. A padding-based defense that reduces the attack accuracy by 25% is also proposed in the paper.

### Brute-Force Detection

A dictionary attack is a form of brute-force attack for breaking an authentication mechanism by trying a huge number of likely possibilities. Lists of the most commonly used passwords are readily available on the Internet, and brute-force tools are usually open-source and shared on GitHub. So even an unskilled attacker can perform dictionary attacks against any web application. Detection of brute-force attacks can be easily done by the web application or by automatic analysis of the server logs. A well-known example of this host-based approach is intrusion prevention software Fail2Ban [35], which scans log files (e.g., `/var/log/auth.log`) and bans IP addresses that show malicious behavior (i.e., too many authentication failures) by adding a rule to the firewall to reject that IP address. This solution, however, is not scalable, does require access to individual servers, and can miss slow attacks that try to stay "under the radar".

On the other hand, a network-level approach is based on analyzing flow data and can be implemented, for example, by an ISP or web hosting provider. Many types of brute-force attacks are *"known to exhibit a characteristic flat* *behavior at the network level, meaning that connections belonging to an attack feature a similar number of packets and bytes, and duration. Flat traffic usually results from repeating similar application-layer actions, such as login attempts in a brute-force attack. For typical attacks, hundreds of attempts span over multiple connections, with each connection containing the same, small number of attempts*" [36]. Many flow-based detection mechanisms were proposed for SSH and HTTP. The paper [37] introduced an intrusion detection system for SSH called SSHCure. It is built on the assumption that a brute-force attack consists of three phases (a scan phase, a brute-force phase, and a die-off phase), and that the traffic in the brute-force phase exhibits characteristic flat behavior. Metrics, such as the number packets per flow or the number of flows per time interval, help to identify attack phases, and eventually detect brute-force attempts and successful compromises.

Network-based detection of brute-force attacks against web applications will be more discussed in Section 2.3.

### 2.1.5 Flow Exporters

#### Joy

Joy [38] is a libpcap-based open-source package for processing live traffic (online mode) and packet capture files (offline mode). It converts the network data into flows represented in a JSON format that contains all the relevant data features. The JSON format is flexible and well-suited for data analysis task and modern programming environments, such as Python and Scikit-learn [39]. Joy can obtain flow data while preserving privacy by avoiding full data capture and by anonymizing IP addresses. In the online mode, raw network data is not retained. It is aware of several important protocols, including HTTP, TLS, DNS, and DHCP, in the sense of being able to store their metadata elements in the output JSON stream. There is also support for extended flow features, such as payload byte distribution and its entropy, and packet-level characteristics. In Joy, the flow key is the conventional 5-tuple, and both unidirectional and bidirectional flows are supported. Apart from the JSON output, Joy can be configured to export flows via IPFIX or Netflow version 9. This functionality, however, is relatively limited, and only the export of the basic fields is supported.

```
{
    "sa": "192.168.0.1",       // IP source address
    "da": "255.255.255.255",   // IP destination address
    "pr": 17,                  // IP protocol number (17 = UDP)
    "sp": 68,                  // UDP source port
    "dp": 67,                  // UDP destination port
    "bytes_out": 900,          // bytes sent from sa to da
    "num_pkts_out": 3,         // packets sent from sa to da
    "time_start": 1479227824,  // start time in seconds since the epoch
    "time_end": 1479227829,    // end time in seconds since the epoch
    "packets": [               // array of packet information
        {
            "b": 300,          // bytes in UDP Data field
            "dir": ">",        // direction: sa -> da
            "ipt": 0           // 0 ms since time_start
        },
        {
            "b": 300,          // bytes in UDP Data field
            "dir": ">",        // direction: sa -> da
            "ipt": 5006        // 5006 ms since last packet
        },
    ],
    "expire_type": "i"
}
```

Figure 2.6: An example JSON format of an unidirectional flow.

### YAF

YAF (Yet Another Flowmeter) was created as a reference implementation of an IPFIX Metering and Exporting Process (described in Section 2.1.1), and to provide a platform for experimentation and rapid deployment of new flow meter capabilities. The authors actively participated in the IPFIX standardization process within the IETF. *"We set out to build a standards-conformant, high-performance, bidirectional network flow meter. Standards-conformance was important to ensure a long operational lifecycle and wide interoperability. The performance was of utmost concern given the scale of the networks we needed to monitor, and the ever-increasing link speeds of the Internet backbone and large enterprise borders. Bidirectionality was important to enable analysis on both sides of communication, as well as to slightly increase export efficiency by eliminating redundant information"* [6]. YAF can export extended information about a large number of protocols via its application labeling functionality. Labeling runs at the flush-and-export time (i.e., once the flow payload is known to be complete). YAF provides two methods for defining labeling rules. ASCII or UTF-8 based protocols, such as SMTP, IMAP, and SIP, lend themselves to textual analysis based on regular expressions. Conversely, for binary protocols, such as DNS, a C-callable plugin can be created. The extraction of packet-level characteristics is not supported. YAF is actively maintained, and the last version was published in March 2019.

## 2.2 Protocols

In this section, I shall briefly review protocols that underlie secure web communication. The focus will be on matters related to the analysis of encrypted web traffic.

### 2.2.1 TLS 1.2

Transport Layer Security (TLS) is a new version of the Secure Sockets Layer (SSL) protocol, which is no longer recommended for use due to its security vulnerabilities. It provides confidentiality, data integrity, non-repudiation, replay protection, and authentication through digital certificates directly on top of the TCP protocol (or any other reliable transport protocol). TLS is currently used for securing the most well-known Internet protocols, such as HTTP, FTP, and SMTP. In this work, I shall focus on the use of TLS within the HTTP protocol that is known as HTTPS, which is probably the most common use of TLS. The TLS protocol consists of two primary components: the TLS Record Protocol and the TLS Handshake Protocol.

*TLS Record Protocol* At the lowest level, layered on top of TCP, is the record protocol that provides connection security with two basic properties:

- The connection is private. Symmetric cryptography is used for data encryption (mostly AES). The keys for this symmetric encryption are generated uniquely for each connection and are based on a secret negotiated by the handshake protocol.
- The connection is reliable. Message transport includes an integrity check using a keyed message authentication code (MAC). Secure hash functions (e.g., SHA-2) are used for MAC computations.

The TLS Record Protocol *"takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients"* [40]. Four TLS subprotocols that sit on top of the record protocol are defined: the handshake protocol, the alert protocol, the change cipher spec protocol, and the application data protocol. Each record has an unencrypted header with content type, protocol version, and payload length. More formally, the TLS record fields are defined as follows:

```
struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;

enum {
    change_cipher_spec (20),
    alert (21),
    handshake (22),
    application_data (23)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length; /* Maximum length is 2**14 (16,384) bytes. */
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

Figure 2.7: A TLS record definition [41].

*TLS Handshake Protocol* The handshake protocol allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits the first byte of data. Handshake messages are supplied to the record protocol and transmitted in the plaintext because no secret keys are yet established. The handshake protocol provides connection security that has three basic properties, which hold even in the face of an attacker who has complete control of the network:

- The peer's identity can be authenticated using asymmetric (public key) cryptography (e.g., RSA, DSA, ECDSA). This authentication can be made optional but is generally required for at least one of the peers. In the case of the HTTPS protocol, usually only the server is authenticated.
- The negotiation of a shared secret is secure: the negotiated secret is unavailable to eavesdroppers. For any authenticated connection, the secret cannot be obtained even by an attacker who can place himself in the middle of the connection.
- The handshake is reliable: no attacker can modify the handshake communication without being detected by the communication parties.

*TLS Application Data Protocol* The application data protocol carries application messages, which are just buffers of data as far as TLS is concerned. The messages are fragmented and encrypted by the record layer, using the current connection security parameters. TLS does not hide the length of the data it transmits. However, newer TLS version 1.3 introduced support for record padding, and endpoints are now able to pad the records in order to obscure lengths and improve protection against traffic analysis techniques [40].

### 2.2.2 TLS 1.3

The new version TLS 1.3 was published in August 2018 nearly ten years after the previous version. The protocol has major improvements in the areas of security, performance, and privacy. It is now supported in most of the libraries and web browsers. Both Mozilla Firefox and Google Chrome use it as default. In contrast to TLS 1.2, TLS 1.3 provides additional privacy for data exchanges by encrypting more of the handshake messages to protect them from eavesdroppers. This enhancement helps protect the identities of the participants and makes traffic analysis less effective. TLS 1.3 also provides forward secrecy by default, which means that the compromise of long term secrets used in the protocol does not allow the decryption of data communicated while those long term secrets were in use [42].

Some of the major differences from TLS 1.2 as listed in the TLS 1.3 specification [40]:

- The list of supported symmetric encryption algorithms has been pruned of all algorithms that are considered legacy. Those that remain are all authenticated encryption with associated data (AEAD) algorithms (e.g., AES-GCM), which combine confidentiality and data integrity into a single cryptographic primitive. Old hash functions, such as MD5, were also removed.

- All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection. Most notably, the server's Certificate handshake message is now encrypted.

- Static Diffie-Hellman and RSA for key exchange have been removed. All public-key based key exchange mechanisms now provide forward secrecy, i.e., they give assurances that session keys will not be compromised even if the private key of the server is compromised.

- The TLS handshake is now completed in one round-trip (1-RTT). An optional zero round-trip time (0-RTT) mode was added, saving a round trip at connection setup for the first application data, at the cost of certain security properties.

Regarding TLS 1.3 and its impact on TLS client fingerprinting, a Cisco researcher explains: *"while more of the TLS handshake goes dark with TLS 1.3, client fingerprinting still provides a reliable way to identify the TLS client. In fact, TLS 1.3 has increased the parameter space of TLS fingerprinting due to the added data features in the ClientHello. While there are currently only five cipher suites defined for TLS 1.3, most TLS clients released in the foreseeable future will be backwards compatible with TLS 1.2 and will therefore offer many "legacy" cipher suites. In addition to the five TLS 1.3-specific cipher suites, there are several new extensions, such as supported versions"* [43]. JA3 authors also state that TLS 1.3 has no effect on the JA3 fingerprint [44].

### TLS 1.3 & Network Visibility

Although TLS 1.3 obscures most of the handshake, including the server certificate, there are several other channels that allow an on-path attacker to determine the domain name the client is trying to connect to, including: (1) cleartext client DNS queries, (2) visible server IP addresses, assuming the server is not doing domain-based virtual hosting, and (3) cleartext Server Name Indication (SNI) in ClientHello handshake messages [45]. DNS over

HTTPS (DoH) and DNS over TLS (DoT) provide mechanisms for clients to conceal DNS lookups from network inspection, and many TLS servers host multiple domains on the same IP address. In such environments, SNI is the only explicit signal used to determine the server's identity. Encrypted SNI (ESNI) specification [45] is currently being developed, and it is already supported in Firefox Nightly and by Cloudflare servers.

An IETF memo called *TLS 1.3 Impact on Network-Based Security* [46] summarizes the impact of TLS 1.3 changes on network visibility from the point of view of enterprises, public sector, and cloud service providers. In these environments, security solutions such as firewalls and intrusion prevention systems rely on some level of network traffic inspection to implement perimeter-based security policies. A traffic monitoring middlebox may, for example, perform vulnerability detection, intrusion detection, crypto audit, compliance monitoring, etc. The following changes were found problematic according to the memo:

- *Encrypted Server Certificate* Organizations may have policies around acceptable ciphers and certificates on their servers. Examples include no use of self-signed certificates, black or white-list Certificate Authority, valid certificate expiration time, etc. In TLS 1.2, the Certificate message was sent in clear-text, however, in TLS 1.3, the message is encrypted, thereby preventing both a network-based auditing and policy enforcement around acceptable server certificates.

- *Removal of Static RSA and Diffie-Hellman Cipher Suites* TLS 1.2 supports static RSA and Diffie-Hellman cipher suites, which enables the server's private key to be shared with server-side middleboxes. TLS 1.3 has removed support for these cipher suites in favor of supporting only ephemeral mode Diffie-Hellman in order to provide forward secrecy. As a result of this, it is no longer possible for a server to share a key with the middlebox a priori, which in turn implies that the middlebox cannot gain access to the TLS session data without being active man-in-the-middle.

- *Version Negotiation and Downgrade Protection* In TLS 1.3, the random value in the ServerHello handshake message includes a special value in the last eight bytes when the server negotiates TLS 1.2 and below. The special value enables a TLS 1.3 client to detect an active attacker launching a downgrade attack when the client did indeed reach a TLS 1.3 server. The primary impact is that TLS 1.3 requires the TLS middlebox to be an active man-in-the-middle from the start of the handshake in order to change these last eight bytes.

### 2.2.3 HTTPS

Hypertext Transfer Protocol (HTTP) is an application protocol that underlies web communication on the Internet since the 1990s. The HTTP is a request/response, text-based, and stateless protocol. A client sends a request to the server in the form of a request method, URI, and HTTP version, followed by a MIME-like message containing request headers, client information, and possible body content over a TCP connection with a server. The server responds with a status line, including the HTTP version and a success or error code, followed by a MIME-like message containing server information, response headers, and possible body content. Three versions of the protocol are used: HTTP/1.0, HTTP/1.1, and HTTP/2. New HTTP/3 specification is being developed. HTTP Secure (HTTPS) is an extension in which the plaintext communication is wrapped in a secure channel provided by TLS.

### 2.2.4 Web Authentication

HTTP cookies are the most common way to do session management and user authentication in web applications. An HTTP cookie is a small piece of data that a server sends to the user's web browser via the Set-Cookie response header. The web browser stores it and sends it back with subsequent requests to the same server in the Cookie request header. This way, the server can determine that the HTTP requests came from the same user. To obtain a session cookie, a user has to provide login credentials. This is usually done by making a POST request to an authentication endpoint with a username and a password in the request body.

```
POST /login HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 42

username=Bob&password=XII1sSM26cy5bN8YQ3wc
```

(a) A simplified web authentication via a POST request.

```
HTTP/1.1 200 OK
Date: Thu, 09 Apr 2020 12:15:53 GMT
Set-Cookie: Session-Id=XihmxFoKQ52YnLrdzHAAXI; Domain=www.example.com
Content-Length: 0
```

(b) A response with a session cookie called Session-Id.

Figure 2.8: An example of a successful HTTP authentication attempt into a web application. Note that without the TLS protection, the user password would be visible in plaintext for every on-path observer.

In case of an incorrect password, the server does not provide a session cookie, and the user has to try again. This can easily be exploited by attack tools that repeatedly try different passwords and check whether the authentication was successful. A common practice is to include a random nonce (also known as CSRF token) in the login form, which has to be sent during authentication and is verified by the server. To perform a brute-force attack, an attacker has to fetch the login page prior to every password attempt to obtain the correct nonce. This, however, is not supported in some available brute-force attack tools, and therefore can deter low-skilled attackers.

## 2.3 Web Applications Brute-Force Attacks

Examples of common web applications are content management systems (CMSs), such as WordPress, Joomla, Drupal, or an e-shop solution Shopify. The popularity of CMSs is underlined by numbers—according to W3Techs [47], WordPress powers 35% of all the websites on the Internet; Joomla, Drupal, and Shopify have around 4% share each. The widespread use of these CMSs also comes with a risk: the fact that anybody can use them, even people with limited technical skills that are unaware of security threats, leads to outdated and vulnerable configurations, and reliance on weak administrator passwords. As such, CMSs end up being a prime brute-force attack target.

In [48], the authors have focused on the defense against brute-force attacks from the perspective of web hosting providers. Three major security threats were identified: (1) brute-force attacks result in an increased load on the underlying infrastructure; (2) following a compromise, malicious scripts can be installed, such as remote access shells; (3) web applications can be misused for a range of illegal activities: distribution of malware, spam campaigns, participation in botnets and DDoS attacks, etc. In such cases, the entire IP space owned by the hosting company may get blacklisted, thus a security mistake made by a single customer potentially impacts all other customers of the hosting company. Detecting attacks against web applications can be done in several ways. Host-based detection protection can be realized, for example, by using CAPTCHA or IP-based authentication blockers. However, such blockers must be implemented and maintained by the customers, who generally have limited technical skills.

### 2.3.1 Network-Based Detection

In this section, a network-based detection mechanism proposed in *Flow-Based Web Application Brute-Force Attack and Compromise Detection* [48] will be discussed in greater detail, because the solution presented in this thesis is inspired by it. The detection approach in [48] is based on unidirectional flow data exported by a flow exporter using IPFIX, and as with many flow-based solutions, the detection operates on flow data chunks that have been received in fixed-length time intervals, typically in the order of several minutes.

*Preselection Phase* The preselection phase serves to make a rough data selection so that the amount of data to be processed in further steps is reduced. To qualify for preselection, a host must have generated at least $N$ flows towards a target server. The threshold $N$ was set to 20 as this value causes benign failed login attempts to be filtered out implicitly, and, on the other hand, reduces the load caused by small attacks and noise. Since the goal of preselection is data filtering and not detection, exceeding the threshold is often easy, even for benign applications, such as web crawlers and calendar fetchers. These two and generally asynchronous requests via AJAX are prone to be misclassified, because, on the network level, they tend to produce a flat traffic pattern similar to a brute-force attack [49]. It is then up the detection phase to classify these cases as benign. The output of the preselection is a list of connection tuples (source and destination IP address, a source port, and a possible hostname) that in the current time window created more than $N$ flow records.

*Detection Phase* The detection is based on packet-level characteristics (discussed in Section 2.1.3), particularly on packet sizes in the form of a histogram. A histogram is created from sizes of all not-empty packets in the flow, with every unique size having its own bin. Empty packets are ignored because they likely are TCP acknowledgments, window updates, or other control information. The next step in identifying brute-force attacks is to aggregate histograms into clusters. Histograms of attack traffic are similar since repeated authentication attempts result in the same packet sizes. As a consequence, the brute-force traffic is clustered. The bottom-up Hierarchical Cluster Analysis (HCA) method is used with Minimum Difference of Pair Assignment (MDPA) as a distance metric, which takes into account the bin sizes. The detection of the brute-force attack is always done using the largest cluster of a connection tuple because it is assumed that attack traffic is dominant enough to comprise a (large) cluster by itself. In cases of non-attack traffic, the largest cluster may contain histograms that are not very similar, meaning that distances between histograms are rather big. To filter out such candidates, the method calculates the average intra-cluster distance for the largest cluster. In case it exceeds a predefined threshold, the connection tuple is labeled as benign. Otherwise, a potential brute-force attack was found. Finally, once the largest cluster is found to feature a brute-force attack, a sanity check to rule out false positives is performed: *"in case the set of clusters features many small clusters, i.e., clusters with only one or two histograms, we overrule the detection. Many small clusters indicate that the network traffic was highly variable in terms of payload, therefore contradicting our definition of typical attack behavior."* [48]

### 2.3.2 Attack Tools

Brute-forcing web application authentication via POST requests is supported in many open-source tools. Usually, a brute-forcer is configured to attack a login endpoint on a specific host. The attacker provides a list of usernames and passwords, the format of the request body, and the request method (usually POST). Moreover, a pattern that can reliably distinguish between failed and successful authentication attempts has to be specified. If the target web application is protected by an access token served as a cookie or a nonce included in a login form, the access token has to be prefetched and extracted via a given regex pattern before every authentication request. The following tools have been studied and used in this thesis:

- Ncrack from the Nmap authors. It has a command-line syntax similar to Nmap and a dynamic engine that can adapt its behavior based on network feedback. A special WordPress module is included. The last release was in August 2019. [50]
- Thc-hydra. Written in C, it can perform rapid dictionary attacks against more than 50 protocols, including HTTP and HTTPS. The last release was in May 2019. [51]
- Patator. A multi-purpose, multi-threaded brute-forcer with a modular design and flexible usage. Written in Python. The last release was in March 2020. [52]

Table 2.2: Feature comparison of open-source brute-force tools.

| | **Custom Headers** | **Conn. Reuse** | **Prefetch Tokens** |
|---|---|---|---|
| Patator | ✓ | ✓ | ✓ |
| Thc-hydra | ✓ | | |
| Ncrack | ✓ | ✓ | |

Chapter 3

# Analysis & Design

## 3.1 Datasets

For the evaluation of my work, I need a representative benign traffic dataset and an HTTPS brute-force attack traffic dataset. A typical approach to this problem is to create your own attacks and mix them with benign, realistic traffic. Freely available datasets usually do not focus on a single specific attack, such as brute-force. The most common attack categories in datasets are DoS attacks, botnet communication, and port scanning. In the case of brute-force attacks, the focus in available datasets is usually on attacks in SSH and FTP. Also, as noted in section 2.3.2, brute-force tools have different behavior patterns (for example, whether connections are reused), and the attack network traffic depends on the target website (which can either use access tokens or not). I need a brute-force dataset that captures this diversity and contains attacks from multiple tools targeted to various websites. I created my own brute-force dataset. Doing it manually would be cumbersome and time-intensive, and would not allow for repeating the same attacks with a different set of parameters. I, therefore, automated the process with a solution based on Docker containers, which is presented in Section 3.1.1. The challenge with the benign traffic dataset was to include enough diversity. As noted earlier, web crawlers, calendar fetchers, and content delivery networks tend to produce similar traffic patterns as brute-force attacks do. It is thus crucial to have such traffic included in the benign dataset, otherwise, any model based on the data would have an unacceptably high number of false positives.

The final dataset consists of three parts:

- Brute-force traffic data generated by the brute-force simulator. This data includes attacks from Patator, Thc-hydra, and Ncrack towards these web applications: WordPress, Joomla, Moodle, Mediawiki, Phpbb, Discourse, and Ghost. It was generated by multiple runs of the brute-force simulator, each with a different set of parameters and various lengths of the password list.

- Traffic from CESNET's backbone network. To obtain the most realistic benign traffic, the data was captured at the perimeter of CESNET's infrastructure in Autumn 2019 and Spring 2020. CESNET fulfills the role of the national research and education network (NREN) within the Czech Republic. Determining that a set of network connections from the backbone network is truly benign is often impossible, and I realize that the backbone traffic could contain a small number of brute-force attacks. However, as long as the proportion of brute-force attacks is minimal, the machine-learning models should cope with those miss-labeled samples (probably by ignoring them as noise). Thus, all backbone traffic is considered as benign.

- StratosphereIPS benign network capture. Stratosphere project [53] published benign HTTPS dataset created by František Střasák. The approach was to browse regular websites, such as Facebook, Twitter, Gmail, and more of the top 500 domains. This dataset contains 13 PCAP files, each around three hours long. The capturing was done by the Firefox browser on Windows 7 running on a virtual machine and by the Iceweasel browser on Kali Linux in April 2017.

### 3.1.1 Brute-Force Generator

In order to generate brute-force traffic data in a repeatable and automatic way, I created a brute-force attack simulator. The main requirement was to support multiple attack tools, various target web applications, and the setup to be as realistic as possible. Target web applications were run in Docker. This way, it is simple to start/stop the applications and to run them one after another. Applications were downloaded from Bitnami's application catalog, which includes popular web applications packaged as Docker containers. The next step was to provide support for TLS communication. TLS is supported in some applications provided by Bitnami, but it would be cumbersome to set up a TLS server or to install a TLS certificate in every Docker container individually. I, therefore, decided to put Nginx as a reverse TLS proxy in front of the Docker containers. The whole setup is presented in Figure 3.1.

The central part is a run script and a remote server with Docker, Nginx, and Tcpdump installed. A valid TLS certificate has to be provided. Target applications are defined in docker-compose files, which include configuration such as required Docker images, environment variables, volumes, and forwarded ports. The applications generally need at least two containers—one for the application itself and one for a database. Fortunately, everything is configured out of the box in the application definitions downloaded from Bitnami. Only the port forwarding had to be changed manually to connect the

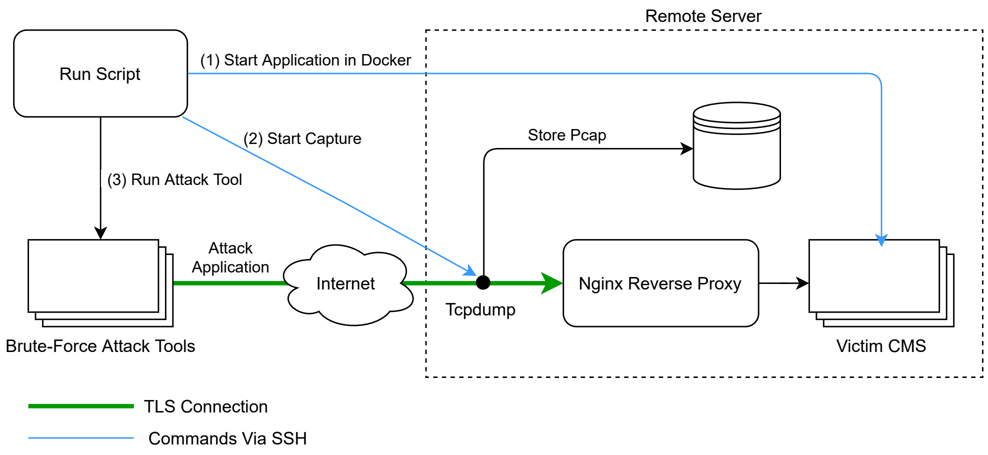


Figure 3.1: The setup of the brute-force attack generator.

application with the reverse proxy properly. Attack tools have to be configured individually for each of the target applications, because they use different HTTP headers and different names for session cookies, and user and password fields. I have used Burp Proxy to determine all required fields for valid authentication. It is crucial to verify that the attack tools are configured correctly, such that the authentication attempts are all valid; otherwise, the captured traffic would not be similar to a realistic attack.

Brute-force traffic data is generated in the following way:

1. The run script connects to the remote server via SSH and starts a docker container with the target application. Docker containers are started/stopped one after another for every defined application. An overview of applications is presented in Table 3.1.

2. For every application, defined attack tools are run against it. Before each attack, a Tcpdump capture is started on the remote server. The result is PCAP files with brute-force traffic, one for every pair of attack tool and target application.

3. Pcaps are downloaded from the remote server.

Table 3.1: An overview of target web applications. When a form authentication is used, the response of an authentication attempt contains the whole HTML document. When AJAX is used, the authentication is performed without reloading the web page, and the response typically contains only a small JSON and a session cookie.e

| **CMS** | **Authentication Method** | **Require Access Tokens** |
|---|---|---|
| WordPress | Form | |
| Joomla | Form | ✓ |
| Moodle | Form | ✓ |
| MediaWiki | Form | ✓ |
| Ghost | AJAX | ✓ |
| Discourse | AJAX | ✓ |
| PhpBB | Form | ✓ |

### 3.1.2 Dataset Structure

Each capture in the dataset contains a Joy output file with flows in the JSON format (one per line) and a readme file with information about the captured traffic, such as the used Tcpdump filter, the date of the capture, and in case of brute-force traffic, the attack tools parameters. The Joy exporter was used because it supports plenty of flow features, and it outputs data in the JSON format, which is well-suited for further data processing with Python.

Every flow in the capture has the following structure:

- Source and destination IPv4 addresses
- Protocol number (e.g., 6 for TCP)
- Source and destination ports
- Time start and time end as unix timestamps
- Number of sent and received bytes
- Number of sent and received packets
- Flow expiration type: ”i” for idle, ”a” for active
- A list of packets contained in the flow; every packet has:
  - Size in bytes
  - Time delay between this and the previous packet in milliseconds
  - Direction of the packet (either ”>” or ”<”)
- A TLS structure with:
  - Client and server TLS versions
  - Server certificate
  - Server name indicator (SNI)
  - Ciphersuits offered by the client and the one selected by the server

  - Client and server extensions
  - Sequence of the record lengths and times (SRLT); every record has:
    * Size in bytes
    * Time delay between this and the previous record in milliseconds
    * Direction of the record (either ">" or "<")
    * Content type (e.g., application data, handshake, alert)
    * A list of handshake messages included in the record (if any), for example Client Hello, Server Hello, or Certificate

## 3.2 Experiments Workflow

I have designed the brute-force detector in an iterative manner as shown in Figure 3.2. By using this workflow, I was able to test and verify various hypotheses, manually analyze data, and feed the insights into improving the preprocessing steps and the dataset.

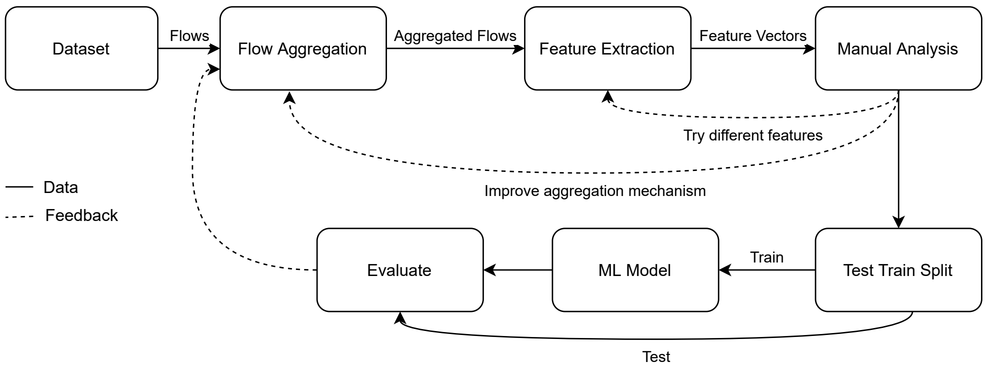


Figure 3.2: Workflow of the experiments.

*Flow Aggregation* After the dataset preparation, it is important to select a suitable representation of samples for machine-learning (ML) algorithms. There are plenty of possibilities, and there is no best option in general, because it depends on the purpose of the algorithm, size of the dataset, and the selected machine-learning algorithm. For example, one ML sample could be: gather all flows with the same source IP, destination IP, destination port, and protocol; or gather all flows going in and out from one IP. Why and how I decided to aggregate flows is summarized in Section 3.3.

*Feature Extraction* The next step is feature extraction. Commonly used flow features were reviewed in Section 2.1.3. I mostly experimented with packet-level characteristics and statistics over them. The results are presented in Section 3.4.

## 3.3 Flow Aggregation

In some applications, it would be possible to consider each flow individually for classifications, but in case of the brute-force detection, some kind of aggregation is needed for the following reasons:

1. Attack tools usually open multiple concurrent connections towards the target, with each having a different source port and thus resulting in a different flow. If flows were considered individually, information about the context of a possible attack (that there are multiple connections) would be lost.

2. Some attack tools, for example, Thc-hydra, open a new connection for every password attempt. Resulting flows look benign as they contain just a few packets. But they form an attack when considered together.

3. Attacks which are longer than the flow active timeout (for example, because they try to "stay under the radar") are split into multiple flows.

For those reasons, the following aggregation mechanism was used. Flows were aggregated by this flow key: source address, destination address, destination port, and TLS server name indicator. The omission of source port causes aggregation of all flows originating from a specific source address towards the target web application. This solves the problem with attack tools like Thc-hydra that open a new connection for every attempt. In contrast with the usual practice, the flow aggregation is not time-based. Instead, flows are being aggregated until the number of packets is bigger than some threshold. This addresses the issue with slower attacks. The pseudo code of the aggregation mechanism is presented in Figure 3.3.

*NAT Problem* The omission of the source port in the aggregation flow key poses one problem. Traffic from different clients that are behind Network Address Translation (NAT) router will be aggregated together. They share the same IP address and the source port, which normally distinguishes them, is ignored in the aggregation. If two hosts that communicate with a target application are both behind the same NAT, and one of them is legitimate, while the other one is launching a brute-force attack, their traffic will be aggregated together, and as a result, the attack might not be detected. This problem could be partially solved by including a TLS client fingerprint (e.g., JA3) in the aggregation key. This way, traffic originating from multiple hosts behind NAT is not aggregated unless the hosts share the same TLS fingerprint (i.e., have the same web browser in the same version, or the same TLS library), which still might happen, but the chance should be lower. Another option would be to include TCP fingerprint.

```
Function is_full(af: aggregated_flow): bool
    return whether the aggregated flow is considered full

Function add_flow(af: aggregated_flow, f: flow): void
    add a flow to the aggregated flow

active_aggregated_flows ← empty hashmap
foreach flow do
    flow_key ← (flow.srcaddr, flow.dstaddr, flow.dstport, flow.sni)
    if flow_key in active_aggregated_flows then
        aggregated_flow ← active_aggregated_flows[flow_key]
        add_flow(aggregated_flow, flow)
        if is_full(aggregated_flow) then
            classify aggregated_flow and remove it from the hashmap
        end
    else
        create new aggregated_flow
        if is_full(aggregated_flow) then
            classify aggregated_flow
        else
            active_aggregated_flows [flow_key ] ← aggregated_flow
        end
    end
end
```

Figure 3.3: Pseudo code of the flow aggregation algorithm.

## 3.4 Features

The purpose of this section is to describe how I proceeded with feature extraction. The goal is to detect brute-force attacks in encrypted traffic. Due to the encryption, no features can be extracted from the payload itself apart from byte distribution and its entropy. However, this has no value in detecting brute-force attacks because both benign and attack traffic are the same encrypted HTTP request, and there should not be differences in the entropy values. On the other hand, packet-level characteristics, such as the sequence of packet lengths, look promising. The underlying idea is that during a brute-force attack, the repeated authentication attempts should exhibit a periodic pattern in network traffic (especially in the sequence of packet sizes).

### 3.4.1 Sequence of TLS Record Lengths

I have decided to use the sizes of TLS records instead of packets. This way, the values are closer to the sizes of HTTP requests and responses (or they are equal to them if the TLS connection is using a cipher in a counter mode). Also, the size of a packet is limited by the maximum transmission unit (typically 1500), which, for example, causes an HTTP response of size 4000 bytes to be split into three packets—1500, 1500, and 1000. When viewed as a TLS record, however, the real size of 4500 bytes could be observed because a TLS record has a maximum size of 16KB. Another advantage is that the sequence of TLS records can be filtered by the content type (handshake, application data, alert). For the task of brute-force detection, only application data records are interesting. Thus only lengths of application data records are considered for feature extraction.

### 3.4.2 Merging of TLS Record Sequences

The next question is how the packet-level features should be extracted from aggregated flows, which combine a number of flows. One option would be to compute packet-level features for every flow individually and then combine them, for example, by averaging. Conversely, one could first merge the sequences of TLS records and only compute the statistics afterward. It is not clear whether or how the merging should be done because the flows in one aggregated flow could in fact be running in parallel. In this work, I decided to sort the flows by their start time and concatenate the record sequences before feature computation. The reasoning is that: (1) Some flows (for example, the case of Thc-hydra traffic) might not contain enough records for calculation of statistics, such as standard deviations and correlation. (2) Computing features over a single merged sequence of records is more efficient. And (3), the error, which is introduced by sequentially concatenating sequences that are running in parallel, is not significant for the used features (mostly means, standard deviations, and correlation). The merging process is illustrated in Figure 3.4.

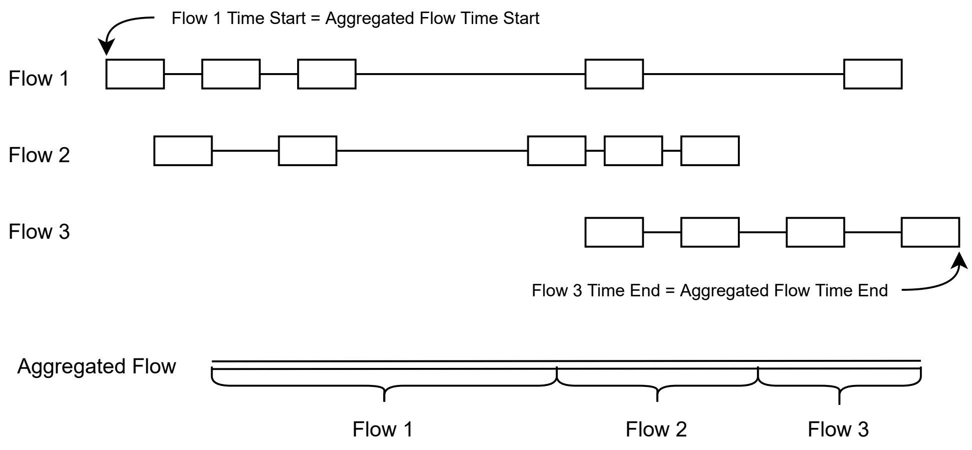

Flow 1 Time Start = Aggregated Flow Time Start
Flow 1
Flow 2
Flow 3
Flow 3 Time End = Aggregated Flow Time End
Aggregated Flow
Flow 1
Flow 2
Flow 3

Figure 3.4: Sequential joining of TLS records in an aggregated flow.

Table 3.2: A summary of selected features.

| **Feature Name** | **Type** |
|---|---|
| Flows Count in the Aggregated Flow | Int |
| Application Data Records Count | Int |
| Time Duration | Int [milliseconds] |
| Mean Flow Duration | Float |
| Bytes Sent | Int |
| Bytes Received | Int |
| Sent/Received Ratio | Float |
| Request Size STD | Float |
| Response Size STD | Float |
| Mean Request Packets | Float |
| Mean Response Packets | Float |
| Roundtrips Count | Int |
| Roundtrips per Second | Float |
| Request Response Autocorrelation | Float |
| Autocorrelation Shift | Int |

One more preprocessing step is performed on the merged sequence of record lengths: consecutive records in the same direction are merged. The result is a sequence of lengths of requests (i.e., out direction, sent) and responses (i.e., in direction, received). An illustration of this step is shown below. The lengths of the received records are represented as negative numbers; the lengths of sent records as positive.

$$\begin{gathered}[300, 250, -2000, -2500, 350, -3000, -3000, -3000, 100, -100] \\ \downarrow \\ [550, -4500, 350, -9000, 100, -100]\end{gathered}$$

The reasoning for merging consecutive packets is that the derived features should help to find patterns of repeated actions (in case of this work, repeated authentication attempts). The lengths of requests and responses are more likely to identify repeated actions rather than the record-level view. As an example, an HTTP request or response could be split into two (or more) TLS records, one for the headers and second (or the rest) for the body. These records are sent one after another, and by summing their sizes, the size of the transmitted HTTP request or response can be obtained.

The final selected features are listed in Table 3.2. The features are:

- **Aggregation Counts**. How many flows were aggregated and the total count of application data records. Note that even though the aggregation is bounded by a threshold of the number of records, the actual number of records can be smaller or bigger than this threshold. It could be bigger because flows are added to the aggregated flow until the number of records is bigger than the threshold, and it could be smaller because, for some aggregation keys, there are not enough flows in the observed traffic to fill the aggregated flow. There are, however, other thresholds in place to skip classification of those aggregated flows, which do not include enough records.

- **Time Duration**. The time span of an aggregation flow starts when the first of its flows starts and ends when the last of its flows ends. The duration is calculated as the difference between those. Also, the average flow duration is computed.

- **Byte Counts**. The total number of bytes sent/received in application records and the ratio of these two values. This ratio can be expected to be around one for authentication attempts done via AJAX requests as the HTTP response contain only a session cookie and thus has a similar size as the POST request. When the authentication is done by a form submission, the ratio is expected to by bigger than one because the HTTP response contains an HTML document of the website, which is much bigger than a POST request. Extreme values of this ratio are indicators of bulk download/upload.

- **Roundtrips**. One request-response pair is called roundtrip. In brute-force traffic, each authentication attempt can be seen as one roundtrip. A request without response could not be an authentication attempt as the attacker would not be able to get to know if the authentication was successful or not. Thus, the number of roundtrips can be seen as the top bound of the number of possible authentication attempts in the communication. Roundtrip count is computed as the minimum of the number of requests and the number responses—these two numbers differ at most by one due to their alternating nature. The number of roundtrips per second is also computed.

- **Request Response Statistics**. Standard deviation is a measure of variation. If the variation of the sizes of either requests or responses is small, it could be an indication of repeated patterns in the communication. Means of response and request sizes are not used because these are specific for every web application (web applications are arbitrarily big), and they do not carry any value in distinguishing between benign and brute-force traffic (legitimate authentication request has the same size as a brute-force password attempt). The average number of packets a request or a response is comprised of (due to the aforementioned merging) is also calculated. The idea is that these could help to distinguish between types of traffic, such as browsing, video streaming, or CDN traffic. As an example, it is unlikely that an HTTP request carrying an authentication attempt is divided into more than, for example, five TLS records. This kind of features could help to reduce the chance of misclassification for traffic types other than browsing.

- **Autocorrelation**. The Pearson correlation coefficient of the request-response sequence with a shifted version of itself is calculated. Note that the sequence is encoded as positive numbers for requests and negative numbers for responses. The purpose is to find a periodic signal in the sequence, which would indicate repeated actions in the communication. The autocorrelation is computed for different lags (how much is the sequence shifted), specifically for lags in the range $[2, Min(Len(X)/3, 10)]$, $Len(X)$ being the sequence length. Lower bound two is chosen because the lag of one would only show whether all values in the sequence are equal (which are not due to the alternating nature of requests and responses). Upper bound is set for performance reasons. The highest autocorrelation value together with its lag are used as features. A brute-force attack on a web application without access token protection can be expected to have high autocorrelation with a lag of two. If an application is protected by an access token, which has to be fetched before every authentication attempt, then the highest autocorrelation is expected to be at the lag of four.

## 3.5 Machine Learning

The task of brute-force detection is a binary classification problem (benign, negative versus brute-force, positive) with a high class imbalance—benign traffic is generally prevalent. Learning a machine-learning model can be either supervised when the dataset is labeled or unsupervised when it is not. The dataset in this thesis is labeled, and thus I shall use the methods of supervised learning. Decision trees were chosen as a starting point because they are one of the most commonly used supervised machine-learning methods for network traffic classification. A decision tree uses a divide-and-conquer approach to solve the classification task by asking a series of questions about the features of the input. Decision trees are relatively efficient to learn and are easy to interpret, i.e., a set of decision rules can be associated with each output. However, they are biased when learned on an unbalanced dataset, which is the case of the dataset in this thesis, where the benign traffic is dominant.

The solution for class imbalance can be either under-sampling (taking a subset) of the majority class, or over-sampling of the minority class, or both combined. When over-sampling, the minority samples can be repeated multiple times, or completely new minority samples can be created. The later is generally preferred, and it is the standard way to tackle the class imbalance. Specifically, a minority over-sampling technique called SMOTE [54] was used to synthetically generate brute-force samples. Note that the generated samples are used only for the training of a model and never for testing. To further improve the classification accuracy, it is common to combine more models together in so-called ensembles. The results from individual models are either averaged (possibly with weights) or a majority vote decides the outcome. I have used a method called adaptive boosting (AdaBoost), which consists of multiple decision trees (or possibly other classifiers) that are learned in a loop. Information gathered at each iteration about the relative hardness of the training samples is fed into the tree learning algorithm such that later trees tend to focus on harder-to-classify samples. The results from individual trees are weighted so that the trees with high accuracy on the training data have a higher weight, and the trees with low accuracy have a lower weight. I have also experimented with an algorithm called LightGBM (Light Gradient Boosting Machine) [55], which is similar to AdaBoost. It also uses multiple decision trees, but the learning is viewed as a numerical optimization problem where the objective is to minimize the loss function of the model by adding more trees.

### 3.5.1 Performance Metrics

In a binary classification task, a tested sample is either: (1) correctly classified as positive, which is called true positive (TP), or (2) correctly classified as negative, which is called true negative (TN), or (3) incorrectly classified as positive when the actual label is negative, which is called false positive (FP), and lastly (4) incorrectly classified as negative when the actual label is positive, which is called false negative (FN). The classification errors, FN and FP, are generally not equal, and it depends on a particular domain to decide, which one is more important. The counts of TP, TN, FP, FN are usually presented in a 2x2 confusion matrix. To transform the confusion matrix into a single score, multiple different measures of classification accuracy exists, but only some are suitable in the presence of class imbalance—recall, precision, false positive rate, and f-score are among the most commonly used.

$$Recall\ (True\ Positive\ Rate) = \frac{TP}{TP + FN}$$

$$False\ Positive\ Rate = \frac{FP}{FP + TN}$$

$$Precision = \frac{TP}{TP + FP}$$

$F_\beta$-score is a measure combining precision and recall such that recall is $\beta$ times more important than precision.

$$F_\beta = (1 + \beta^2) \cdot \frac{Precision \cdot Recall}{(\beta^2 \cdot Precision) + Recall}$$

Two commonly used values for $\beta$ are 2, which weighs recall higher than precision (by placing more emphasis on false negatives), and 0.5, which weighs precision higher than recall (by placing more emphasis on false positives). Due to the scale of monitored network traffic and the expensive remediation actions (e.g., investigating a possible incident), maintaining a low false positive rate is more important. Therefore, I used the $F_{0.5}$ score as the metric for the evaluation of the machine-learning models.

Chapter 4

# Implementation

## 4.1 Implementation of NEMEA Module

The brute-force detector was implemented as a detection module into a system for network traffic analysis and anomaly detection called NEMEA [56]. The final production architecture is presented in the following Figure 4.1. In this section, I will discuss how the individual parts are implemented and configured.

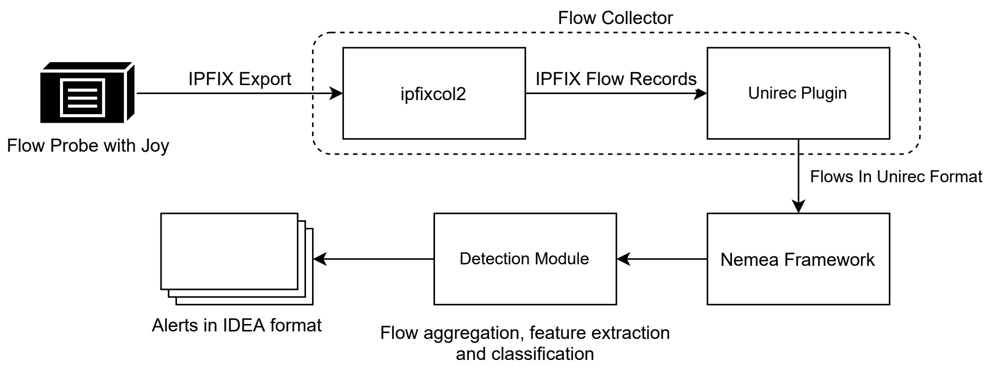


Figure 4.1: The NEMEA brute-force detector setup.

### 4.1.1 Joy IPFIX Exporter

Even though Joy supports IPFIX export, the number of fields that can be exported was fairly limited. Namely, Joy exported only source and destination IPv4 addresses, ports, and start and end times. Joy is written in C and the source code is hosted on Github. I extended the IPFIX exporting functionality to include all fields that are needed by the detection module in order to compute all machine-learning features. The changes were merged into CESNET's fork of Joy.

Table 4.1: Summary of added IPFIX information elements. Private Enterprise Numbers (PEN) are used to define IPFIX elements outside of the IANA registry.

| **Information Element** | **PEN** | **Id** |
|---|---|---|
| IPFIX_PPI_TLS_REC_LENGTHS | 8057 (CESNET) | 1010 |
| IPFIX_PPI_TLS_REC_TIMES | 8057 | 1011 |
| IPFIX_PPI_TLS_CONTENT_TYPES | 8057 | 1012 |
| IPFIX_PPI_PKT_LENGTHS | 8057 | 1013 |
| IPFIX_PPI_PKT_TIMES | 8057 | 1014 |
| IPFIX_PPI_PKT_FLAGS | 8057 | 1015 |
| IPFIX_PPI_PKT_DIRECTIONS | 8057 | 1016 |
| IPFIX_TLS_SNI | 39499 (Flowmon) | 338 |

A new IPFIX template `extended_template` was defined, which includes standard IEs (source and destination addresses, ports, start and end times) together with newly added IEs listed in Table 4.1. All added fields (except IPFIX_TLS_SNI) are implemented as IPFIX structured elements (basicLists) of variable length. Joy can be run in an exporter mode with the `extended_template` by using these command-line arguments:

- `ipfix_export_template=extended`. To use an extended IPFIX template with the new information elements.
- `tls=1`, `bidir=1`, `ppi=1`. To report TLS session metadata, to use bidirectional flows, and to report TCP per-packet information.
- `ipfix_export_remote_port`, `ipfix_export_remote_host`, and `ipfix_export_port`. To configure IPFIX collector address and port and the local exporter port.

### 4.1.2 IPFIX Collector IPFIXcol2

IPFIXcol2 is a high-performance Netflow and IPFIX collector developed by CESNET [57]. The key features are support for bidirectional flows and support for structured data types (e.g., basicLists). It is able to output the flow data in an UniRec format, which is the native format of the NEMEA system. The collector: (1) listens on a specified port and receives IPFIX message, (2) processes the flow records and converts IEs to UniRec fields according to the mappings listed in Table 4.2, and (3) outputs the flow data to a specified interface, from where the data is processed by NEMEA detection modules.

Table 4.2: IPFIX to UniRec mappings. For example, e0id291/e8057id1010 means that basicList (id 291) of IPFIX_PPI_TLS_REC_LENGTHS (specified in 4.1) elements should be converted into an UniRec field with name PPI_TLS_REC_LENGTHS and type array of int16.

| **UniRec Name** | **UniRec Type** | **IPFIX IEs** |
|---|---|---|
| PPI_TLS_REC_LENGTHS | int16* | e0id291/e8057id1010 |
| PPI_TLS_REC_TIMES | unt16* | e0id291/e8057id1011 |
| PPI_TLS_CONTENT_TYPES | uint8* | e0id291/e8057id1012 |
| PPI_PKT_LENGTHS | int16* | e0id291/e8057id1013 |
| PPI_PKT_TIMES | uint16* | e0id291/e8057id1014 |
| PPI_PKT_FLAGS | int8* | e0id291/e8057id1015 |
| PPI_PKT_DIRECTIONS | int8* | e0id291/e8057id1016 |
| TLS_SNI | string | e339499id338 |

### 4.1.3 Brute-Force Detection Module

I have decided to implement the brute-force detection module in Python 3. The core functionality of the module can be summarized as follows. On initialization, the module loads a learned classifier from a file in the Pickle format, which is a protocol for serializing and de-serializing Python objects. Then, flow aggregation is continuously performed on received data as described in Section 3.3. The resulting aggregated flows are stored in a table implemented as a Python dictionary. To keep the table in a managable size, a separate thread is run every $N$ seconds that timeouts all records that are inactive for more then $M$ seconds. When an aggregated flow is full or is timeouted, the `predict` method of the classifier is called to decide whether the sample is brute-force or not. In both cases, the record is removed from the table.

The input of flow data and output of the alerts is done via the Pytrap library, which implements communication interfaces for NEMEA modules. It ensures that the data on the input has the required set of fields, and throws an exception otherwise.

The expected UniRec fields for the brute-force detection module:

- ipaddr `SRC_IP` - the source address.
- ipaddr `DST_IP` - the destination address.
- uint16 `SRC_PORT` - the source port.
- uint16 `DST_PORT` - the destination port.
- uint8 `PROTOCOL` - the protocol identifier.

- time TIME_FIRST - the flow start time.
- time TIME_LAST - the flow end time.
- int16* PPI_TLS_REC_LENGTHS - an array of TLS record lengths. The directions of the records is encoded as a negative number for received records, and as a positive number for sent records.
- uint8* PPI_TLS_CONTENT_TYPES - an array of TLS record content types.
- string TLS_SNI - the server name indicator (i.e., domain).

The output format for brute-force alerts:

- ipaddr SRC_IP - the attacker source address.
- ipaddr DST_IP - the attack destination address.
- uint16 DST_PORT - the attack destination port.
- string TLS_SNI - the target server name indicator.
- time EVENT_TIME_START - the event start time is the aggregated flow start time.
- time EVENT_TIME_END - the event end time is the aggregated flow end time.

### Configuration

The available command-line arguments of the detection module.

- -i, --ifspec. Configure the input interface, from where the module should read flow data, and the output interface where the module should send alerts; separated by comma.
- -v, --verbose. Tell the module to output debug information, such as the current number of aggregated flows in the table.
- -c, --classifier. A pickle file from which to read the classifier, the default is `classifier.pickle`.
- --idle_timeout. After how many seconds should an aggregated flow be considered inactive and thus be classified and removed from the table. The default is 180 seconds.
- --timeout_interval. How often should a separate thread run to timeout idle aggregated flows. The default is 60 seconds.

### Alerts in IDEA Format

When a possible brute-force attack is detected, an alert is created and sent to the output interface, which also use the UniRec format. These alerts are then converted to an Intrusion Detection Extensible Alert (IDEA) format, which is a JSON format for intrusion detection alerts and other security events [58]. This is the final output of the brute-force detection module.

```
{
    "Format": "IDEA0",
    "ID": "07775df5-6ee3-4e5f-a694-8b3212768856",
    "DetectTime": "2020-04-26T14:36:39Z",
    "CreateTime": "2020-04-26T14:36:39Z",
    "EventTime": "2020-04-17T08:19:21Z",
    "CeaseTime": "2020-04-17T08:19:48Z",
    "Category": ["Attempt.Login"],
    "Description": "Possible bruteforce attack in HTTPS.",
    "Note": "Detected a traffic pattern similar to a bruteforce
             attack. Detection is based on per packet information,
             such as packet sizes and inter-arrival times.",
    "Source": [
        {
            "Proto": ["tls"],
            "IP4": ["80.112.137.128"]
        }
    ],
    "Target": [
        {
        "Proto": ["tls"],
        "Port": [443],
        "Hostname": ["dev.luxemburk.com"],
        "IP4": ["172.105.77.10"]
    }
    ],
    "Node": [
        {
            "Name": "com.example.nemea.https_bruteforce_detector",
            "SW": ["Nemea", "https_bruteforce_detector"],
            "Type": ["Flow", "Statistical"]
        }
    ]
}
```

Figure 4.2: An example of an alert about a possible brute-force attack in the IDEA format.

## 4.2 Implementation of Brute-Force Dataset Generator

The brute-force generator is written in Python 3. The remote server, where the Docker containers run and the traffic is captured, is an Ubuntu 18.04 with Nginx, Docker, and Tcpdump installed. A TLS certificate was obtained from the Let's Encrypt certificate authority, which offers free certificates and provides the Certbot tool that simplifies the configuration of Nginx as a TLS server and automates the certificate renewal. The run script authenticates to the remote server by an SSH key, which was installed in advance. Also, all commands to the remote server, such as the start of a Docker container or the start of a Tcpdump capture, are performed via the SSH protocol, which is implemented by a Python library called Paramiko.

### Configuration

Attack tools are configured in the run script as Python dictionaries with f-strings used to substitute variables such as the target host, the username that should be attacked, the password list (with one password per line; this define how big the attack is), and the number of threads. I have mostly run the generator with password lists of size 2000. An example of the WordPress attack configuration is shown in Figure 4.3.

```
"wordpress": {
  "login_url": "/wp-login.php",
  "attacks": {
    "hydra": f"""
      {HYDRA_PATH} -I -S -T {THREADS} -l {TARGET_USER}
      -P {PASSWORDS_FILE} {HOST}
      https-post-form /wp-login.php:log=^USER^&pwd=^PASS^:incorrect""",
    "ncrack": f"""
      {NCRACK_PATH} -P {PASSWORDS_FILE}
      --user {TARGET_USER} wordpress-tls://{HOST}""",
    "patator": f"""
      {PATATOR_PATH} http_fuzz url=https://{HOST}/wp-login.php
      method=POST body="log={TARGET_USER}&pwd=FILE0" 0={PASSWORDS_FILE}
      -x ignore:code=200 --threads {THREADS} persistent={PERSISTENT}""",
}
```

Figure 4.3: The configuration of Thc-hydra, Ncrack, and Patator attack tools against WordPress.

## 4.3 Implementation of Feature Extraction and Machine-Learning

The ML part was also written in Python 3 as an interactive notebook, using standard libraries such as Numpy, Pandas, and Scikit-learn [39]. For decision trees, Scikit-learn implements an optimized version of the CART [59] algorithm. For the adaptive boosting ensemble it implements the AdaBoost-SAMME.R algorithm [60]. The main process of fitting (i.e., learning) a classifier and of doing experiments can be divided into several tasks:

1. Load the dataset, which is divided into two directories: `normal` and `bruteforce`. The data saved on disk is in the form of gunziped JSON files. It has to be loaded, aggregated, and converted to feature vectors. The method for extraction of the features is shown in an abbreviated form in Figure 4.4. The resulting Pandas DataFrame (i.e., table) is saved to disk so that it can be later loaded without the need to re-create it, which is only done when new features are added.

2. The standard next step is to divide the data into two parts: train data and test data. The train subset is used to learn the model, which is then evaluated on the test subset. This way, the performance can be measured on data that the model has not seen during the learning. A more robust approach is called cross validation: the data is split into N parts, and the model is learned N times. In each iteration, the model is trained with N-1 subsets and tested on the one that is left out. After the last iteration, each subset was used exactly once for testing. The class predictions from each iteration are merged together to compute the final score—this is preferable to computing the score in each iteration and averaging them because that introduces bias [61]. Stratified cross validation ensures that the proportion of brute-force and benign samples is the same in each split, which is needed for accurate scoring in the presence of class imbalance. An issue related to the standard cross validation and the dataset presented in this work is discussed in Section 5.1.

3. Visualizing the results is necessary to get meaningful insights into how and why a model makes decisions. Some classifiers, for example a single decision tree, can have their whole internal structure plotted to show the decision boundaries and important features. With ensemble models, this becomes impractical as the number of classifiers can be in the order of tens or hundreds. In all cases, a plot of a confusion matrix with the distribution of TP, TN, FP, FN is helpful. For the figures in the Evaluation Chapter, I have mostly used the Matplotlib library and Scikit-learn plotting functions.

```
def to_feature_dict(self):
  time_duration = self.time_end - self.time_start
  app_packets = merge_app_packets(self.flow_list)
  d = {
    "sa": self.key[0],
    "da": self.key[1],
    "dp": self.key[2],
    "tls_fingerprint": self.key[3],
    "sni": self.key[4],
    "time_duration": time_duration,
    "num_app_pkts": self.packets_count,
    "label": self.label,
    "app_packets": app_packets,
    "no_flows": len(self.flow_list),
    FLAG_BENIGN: None,
    "time_start": self.time_start,
    "time_end": self.time_end,
    "mean_flow_duration":
        np.mean(list(map(lambda f: f.time_duration, self.flow_list))),
  }
  if self.packets_count < APP_PKTS_MIN:
    d[FLAG_BENIGN] = BenignReason.NOT_ENOUGH_APP_PACKETS
    return d
  rr_sizes, req_sizes, resp_sizes, req_pkt_counts, resp_pkt_counts \
      = get_requests_responses(app_packets)
  roundtrips = min(len(request_sizes), len(response_sizes))
  d["roundtrips"] = roundtrips
  if roundtrips < ROUNDTRIP_MIN:
    d[FLAG_BENIGN] = BenignReason.NOT_ENOUGH_ROUNDTRIPS
    return d
  d["request_response_sizes"] = rr_sizes
  d["app_bytes_out"] = sum([p for p in app_packets if p > 0])
  d["app_bytes_in"] = abs(sum([p for p in app_packets if p < 0]))
  d["in_out_ratio"] = d["app_bytes_in"] / d["app_bytes_out"]
  d["request_sizes_std"] = np.std(req_sizes)
  d["response_sizes_std"] = np.std(resp_sizes)
  d["request_pkts_mean"] = np.mean(req_pkt_counts)
  d["response_pkts_mean"] = np.mean(resp_pkt_counts)
  d["request_pkts_std"] = np.std(req_pkt_counts)
  d["response_pkts_std"] = np.std(resp_pkt_counts)
  d["roundtrips_per_sec"] = roundtrips / time_duration
  d["rr_autocorr"], d["rr_lag"] = compute_autocorrelation(rr_sizes)
  d["rr_periods"] = d["num_app_pkts"] // d["rr_lag"]
  d["packets_per_flow"] = d["num_app_pkts"] / d["no_flows"]
  return d
```

Figure 4.4: The feature extraction method that converts an aggregated flow to a dictionary, which is then loaded into Pandas DataFrame.

Chapter 5

# Evaluation

This Chapter presents an evaluation of the brute-force attack detection technique. For the measurements, I have used the dataset described in Section 3.1. The size of the final dataset and the distribution of benign/brute-force samples is shown in Table 5.1. The numbers of brute-force samples from different attack tools are shown in Table 5.2. The class imbalance in the dataset is one brute-force sample for every 75 benign samples.

Table 5.1: The number of flows and aggregated flows in the dataset. The brute-force samples (aggregated flows) accounted for 1.3% of all samples (1:75 ratio).

| | | **Flows** | | **Aggregated Flows** | |
|---|---|---|---|---|---|
| | | Benign | Brute-Force | Benign | Brute-Force |
| **Attacks** | WordPress | | 10 818 | | 494 |
| | Joomla | | 2 763 | | 209 |
| | Moodle | | 584 | | 65 |
| | MediaWiki | | 1 140 | | 22 |
| | PhpBB | | 828 | | 36 |
| | Ghost | | 2 064 | | 97 |
| | Discourse | | 319 | | 21 |
| | **CESNET** | 1 709 106 | | 67 007 | |
| | **P. Stratosphere** | 256 539 | | 2 444 | |
| | **Total** | 1 965 645 | 18 516 | **69 451** | **944** |

Table 5.2: The distribution of brute-force samples by the used attack tool.

| **Attack Tool** | Patator | Ncrack | Thc-hydra |
|---|---|---|---|
| **Aggregated Flows** | 570 | 57 | 317 |

## 5.1 Cross Validation Setup

After initial experiments, I have discovered a problem with the brute-force part of the dataset. The samples created with one configuration of an attack tool and a target web application (e.g., Patator and WordPress) are sometimes too similar to each other. Such samples are called near duplicates or twins. The result is that when the data is split to test and train subsets, there is a chance that almost identical samples end up in both train and test data, and as a consequence, the classifier exhibits a surprisingly high accuracy because it is tested on data that it has already seen during learning. To overcome this issue, I have devised a new approach to splitting the data during the cross validation. The brute-force data is divided into eleven groups according to the used attack tool, web application, and whether connections were reused. Examples are: `joomla_patator`, `wordpress_ncrack`, `ghost_no_persistent_patator`, etc. Then, in each iteration of the cross validation, two groups of the brute-force data are held out for testing, and the normal data is randomly divided with 0.3 test ratio. There is an iteration for every combination of 2 from 11 brute-force groups, which is 55 iterations in total. This new splitting technique models a more realistic scenario when the classifier has to label traffic from a previously unseen attack tool or against a previously unseen web application. The classifiers are expected to perform worse when evaluated with the leave-groups-out cross validation, but that is favorable as it gives a space for improvement (for example, by using better/more features).

## 5.2 Classification

To establish a base-line performance, I have measured three models: logistic regression, k-nearest neighbors (KNN), and a single decision tree. The parameters of the models were not optimized. The results are presented in Table 5.3. Only the decision tree was able to recognize a considerable portion of brute-force samples while keeping the number of false positives moderately low, but still unacceptably high for network monitoring purposes.

Table 5.3: The base-line performance of three basic classifiers. The KNN considered 10 nearest neighbors. The maximum depth of the decision tree was set to 5, and the tree learning algorithm was further constrained with a minimal split impurity decrease parameter. Other parameters were the defaults of Scikit-learn. The decision tree is shown in Figure 5.1.

| | $F_{0.5}$ **Score** | **Recall** | **False Positive Rate** |
|---|---|---|---|
| **Logistic Regression** | 0.242 | 0.879 | 0.028 (1:35) |
| **Decision Tree** | 0.754 | 0.719 | 0.002 (1:500) |
| **KNN** | 0.280 | 0.232 | 0.004 (1:250) |

### 5.2.1 Ensemble Methods

Next, I evaluated ensemble methods: AdaBoost and LightGBM (implemented in Scikit-learn as class `HistGradientBoostingClassifier`). I have experimented with multiple ameters with the goal of finding the most performant classifier in the leave-groups-out cross validation measured by $F_{0.5}$ score and false positive rate. The analyzed hyperparameters, their defaults in Scikit-learn, and the tested values, are shown in Table 5.4 for AdaBoost and in Table 5.5 for LightGBM.

Table 5.4: Hyperparameters of AdaBoost with a decision tree as a classifier. The tree max features parameter defines the number of features to consider when looking for the best split during the learning of a tree (note that the features are randomly shuffled before taking sqrt/log2 of them). The criterion is a function to measure the quality of a split; options are either information gain (i.e., the decrease of the entropy after a split), or Gini impurity.

| | **Default** | **Tested** |
|---|---|---|
| **Number of Estimators** | 50 | 50,75,100,125,150 |
| **Tree Max Depth** | None | 3,5,8 |
| **Tree Max Features** | All | Log2,Sqrt,All |
| **Tree Split Criterion** | Gini | Gini,Entropy |
| **SMOTE Sampling Ratio** | 1 | 0.5,0.7,1 |

Table 5.5: Hyperparameters of the LightGBM model. Before training, each feature is binned into integer-valued bins. The number of bins is limited by the max bins parameter; there could be at most 255 bins. The learning rate parameter is set to prevent over-fitting by slowing down the learning process.

| | **Default** | **Tested** |
|---|---|---|
| **Maximum Iterations** | 100 | 100,150,175,200 |
| **Tree Max Depth** | None | 3,5,8 |
| **Learning Rate** | 0.1 | 0.01,0.1,0.2 |
| **Max Bins** | 255 | 50,100,150,200,255 |
| **SMOTE Sampling Ratio** | 1 | 0.5,0.7,1 |

For both classifiers, the best SMOTE ratio was one, i.e., over-sample so that the number of both classes is the same. The best AdaBoost peformance was measured with 75 trees having the max depth of 3, considering square root of the number of features at each split. The criterion function has no impact on the performance. The best LightGBM performance was measured with: maximum 175 iterations, max tree depth of 3, learning rate at its default value 0.1, and with 50 maximum bins.

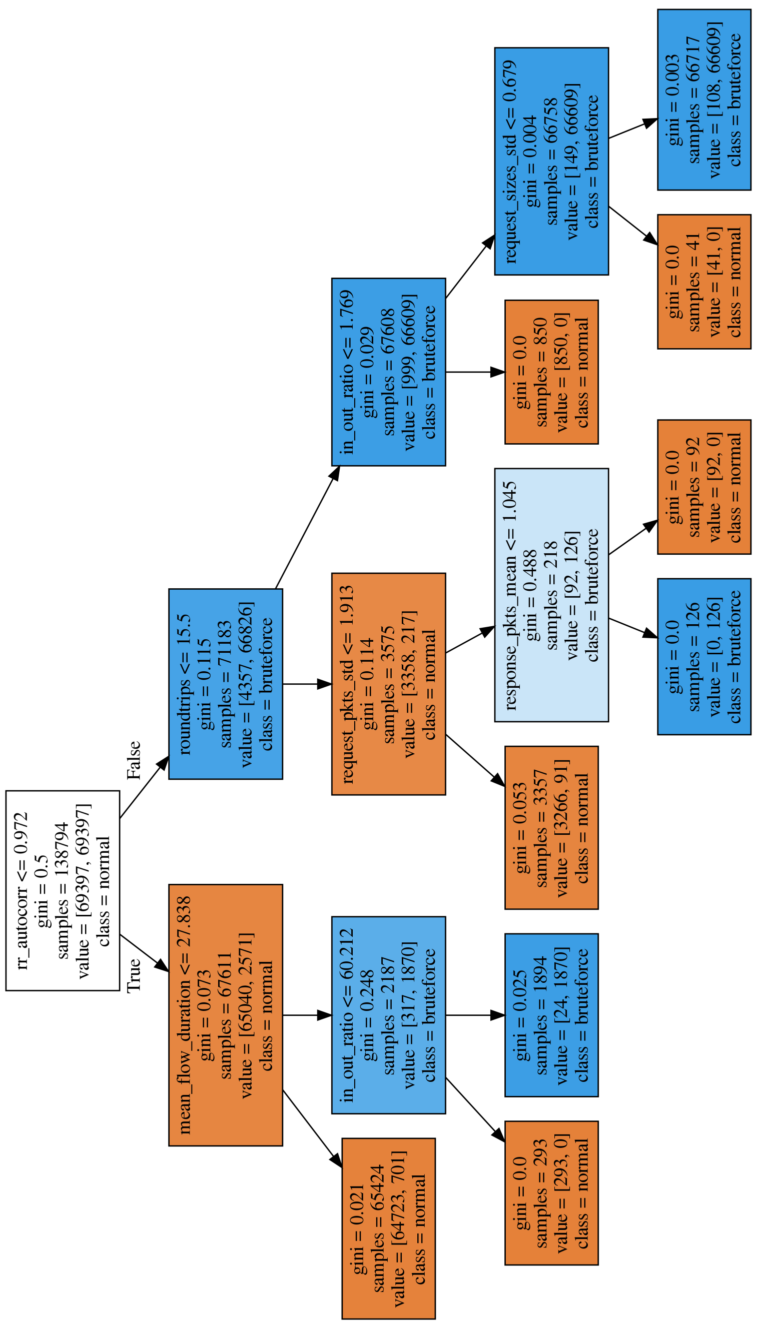


Figure 5.1: Learned decision boundaries of the baseline decision tree model. The nodes with brute-force majority are colored blue; the nodes with benign majority are colored orange. The hue of the color represents how strong the majority is, i.e., white means half and half split with no majority. The root node is white because SMOTE was used to over-sample brute-force data to a one-to-one ratio.

Table 5.6: The performance of optimized AdaBoost and LightGBM compared to the base-line decision tree.

| | $F_{0.5}$ **Score** | **Recall** | **False Positive Rate** |
|---|---|---|---|
| **Base-line Tree** | 0.754 | 0.719 | 0.002 (1:500) |
| **AdaBoost** | 0.902 | 0.723 | 0.00004 (1:25000) |
| **LightGBM** | 0.953 | 0.844 | 0.0001 (1:10000) |

### 5.2.2 Results

The ensemble methods performed substantially better than the best base-line classifier, the single decision tree. The final results are shown in Table 5.6. The $F_{0.5}$ score raised from 0.75 to 0.9 with AdaBoost and to 0.95 with LightGBM. The main improvement, however, is in the false positive rate, which lowered from 1:500 ratio to 1:25000 with AdaBoost and to 1:10000 with LightGBM.

The LightGBM classifier has the best recall of 0.84, and I would consider it as the best brute-force classifier with the right balance between the number of detected brute-force attacks and the number of false positives. The exact numbers of false positives, false negatives, etc., are shown in a confusion matrix in Figure 5.2.

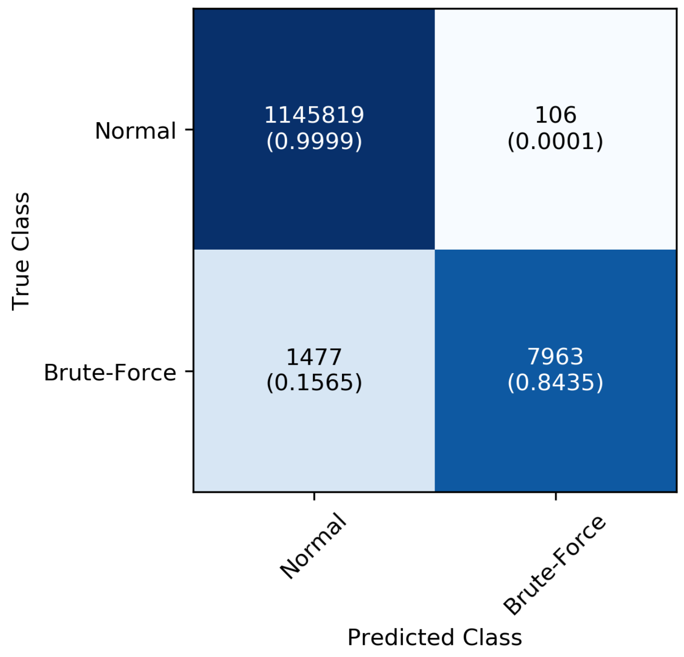


Figure 5.2: The confusion matrix of the optimized LightGBM classifier. The matrix was summed across all splits of the leave-groups-out cross validation.

## 5.3 Discussion

### 5.3.1 Feature Importances

After the AdaBoost classifier is learned on the train data, it provides statistics about how much individual features are useful for the classification task. The importance of a feature is measured as the total information gain of tree nodes that use that feature for splitting the data. The feature importances are averaged from each cross validation split and are rescaled such that their sum is one. The final feature importances are shown in Figure 5.3. Each feature was created based on a specific idea of why and how it should help in the brute-force detection task as described in Section 3.4. Next, I would like to review whether the most important features are those as expected and whether their significance can be explained.

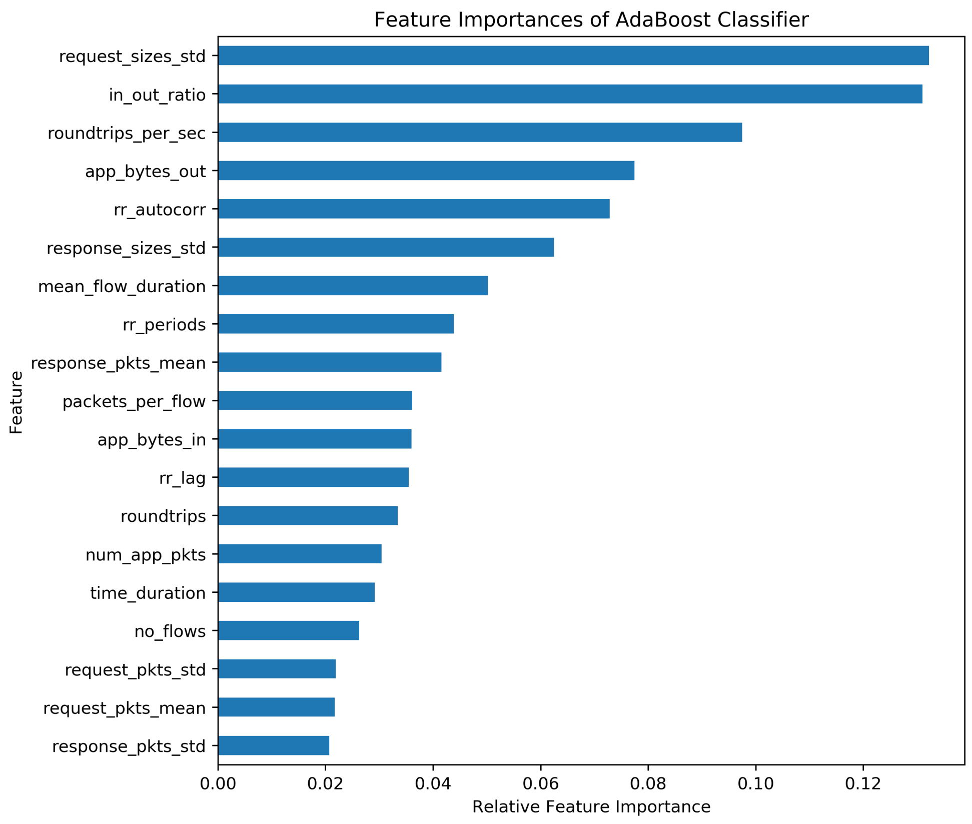


Figure 5.3: Relative importances of features learned by the AdaBoost classifier.

The top five features are: (1) the standard deviation of request sizes, (2) the received/sent ratio, (3) the number of roundtrips per second, (4) the sum of sent bytes, and (5) the autocorrelation of the request-response sequence.

A low standard deviation of request sizes is a sign of repeated user actions (e.g., authentication attempts), which could constitute a brute-force attack. Received/sent ratio helps to distinguish between different network activities, such as mostly download/upload or balanced. Having the received/sent ratio smaller than, for example, 0.5 (i.e., 2 times more sent than received), is improbable for any web application authentication mechanism. Therefore, such traffic can be quickly labeled as benign. The number of roundtrips per second gives the upper bound of the frequency of password attempts, i.e., when this number is moderately low, the potential attack would be slow and ineffective (however, an attacker can arbitrarily slow the attack in order to prevent detection). The autocorrelation helps to find periodic patterns in the sequence of requests and responses, which are expected in brute-force traffic because one action is repeated over and over. Overall, the order of features have met the expectations and the importance of the top features is clearly interpenetratable.

### 5.3.2 Future Work

In this section, I would like to discuss how future work could improve and extend the solution proposed in this thesis. As a next step, I would propose to design and implement a system for validation of the brute-force detection solution on real-world data in a production environment. However, in order to validate the results in production, ground truth labels are needed, and it is not clear how to get them. One possible approach could be like this: (1) get access to a set of production web servers running WordPress, (2) install WordPress Fail2ban plugin that logs all successful and unsuccessful login attempts (3), configure Fail2ban to send alerts to a remote server, (4) deploy the detection module proposed in this thesis such that it could monitor all network traffic towards the web servers, and finally (5) compare the alerts generated by the detection module and by Fail2ban. This way, it would be possible to compare the network level detection with ground truth from web server logs.

Future research could also investigate if TLS record sizes can be replaced with packet sizes for feature computation without losing much detection performance. This would simplify the requirements for the used flow exporters. Also, limiting the number of packets that are captured in each flow could further reduce the workload of the flow exporters while possibly keeping the same detection performance.

# Conclusion

The detection of brute-force attacks against web applications is mostly performed at the host level with access to the log files. However, host-based detectors come with drawbacks; most notably, they require access to the protected servers, and they are cumbersome to deploy and maintain at scale. This thesis proposes a new brute-force detection method that is applicable at the network level. The advantages of the method are that it does not require access to the log files, is more scalable, is based on open standards such as IPFIX, and most importantly, it works with encrypted HTTPS traffic. More and more network traffic is being encrypted, and it is crucial to update our intrusion detection methods in order to maintain at least some level of network visibility. The proposed solution could be particularly useful for web hosting providers, who would like to protect their customers against brute-force attacks without the need to have access to individual servers. Also, an organization could use the method to detect outgoing brute-force attacks, which would indicate a compromised device in the network of the organization.

The thesis reviewed the current state of the art in brute-force detection and summarized the theoretical background needed for creating a functional brute-force detector.

As part of the thesis, a new dataset was created that consists of benign HTTPS traffic from CESNET's backbone network and brute-force traffic generated with open-source attack tools and popular web applications run in Docker. The dataset is publicly available and will be used in future research.

The thesis described a design of a brute-force attack detection method, which is based on packet-level characteristics, such as packet sizes, and on a novel flow aggregation approach specialized for the detection of brute-force attacks coming from different tools. The best detection performance was achieved with the LightGBM model, which discovered 84% of attacks with a false positive rate of 1:10 000. Last but not least, documentation and implementation of a detector module for the NEMEA system was provided.

# Bibliography

1. *Let's Encrypt Stats - Let's Encrypt - Free SSL/TLS Certificates* [online] [visited on 2020-05-27]. Available from: `https://letsencrypt.org/stats/`.
2. SANDERS, Chris; SMITH, Jason. *Applied Network Security Monitoring: Collection, Detection, and Analysis.* Syngress, 2013. ISBN 978-0-12-417208-1.
3. TRAMMELL, Brian; CLAISE, Benoit. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information* [online] [visited on 2020-02-16]. Available from: `https://tools.ietf.org/html/rfc7011`.
4. HOFSTEDE, Rick; CELEDA, Pavel; TRAMMELL, Brian; DRAGO, Idilio; SADRE, Ramin; SPEROTTO, Anna; PRAS, Aiko. Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *IEEE Communications Surveys & Tutorials* [online]. 0024–2014, vol. 16, no. 4, pp. 2037–2064 [visited on 2020-02-11]. ISSN 1553-877X. Available from DOI: `10.1109/COMST.2014.2321898`.
5. ZHANG, Jian; MOORE, Andrew. Traffic Trace Artifacts Due to Monitoring Via Port Mirroring. In: *2007 Workshop on End-to-End Monitoring Techniques and Services.* 2007, pp. 1–8. Available from DOI: `10.1109/E2EMON.2007.375317`.
6. INACIO, Chris; TRAMMELL, Brian. YAF: Yet Another Flowmeter.
7. TRAMMELL, Brian H.; BOSCHI, Elisa. *Bidirectional Flow Export Using IP Flow Information Export (IPFIX)* [online] [visited on 2020-02-25]. Available from: `https://tools.ietf.org/html/rfc5103`.
8. WAGNER, Arno; TRAMMELL, Brian; CLAISE, Benoit. *Flow Aggregation for the IP Flow Information Export (IPFIX) Protocol* [online] [visited on 2020-02-21]. Available from: `https://tools.ietf.org/html/rfc7015`.

9. TRAMMELL, Brian; BOSCHI, Elisa. *IP Flow Anonymization Support* [online] [visited on 2020-02-21]. Available from: `https://tools.ietf.org/html/rfc6235`.
10. TRAMMELL, Brian; BOSCHI, Elisa. An Introduction to IP Flow Information Export (IPFIX). *IEEE Communications Magazine*. 2011, vol. 49, no. 4, pp. 89–95. ISSN 1558-1896. Available from DOI: `10.1109/MCOM.2011.5741152`.
11. *IP Flow Information Export (IPFIX) Entities* [online] [visited on 2020-02-26]. Available from: `https://www.iana.org/assignments/ipfix/ipfix.xhtml`.
12. TRAMMELL, Brian; CLAISE, Benoit. *Information Model for IP Flow Information Export (IPFIX)* [online] [visited on 2020-02-26]. Available from: `https://tools.ietf.org/html/rfc7012`.
13. DHANDAPANI, Gowri; AITKEN, Paul; YATES, Stan; CLAISE, Benoit. *Export of Structured Data in IP Flow Information Export (IPFIX)* [online] [visited on 2020-03-11]. Available from: `https://tools.ietf.org/html/rfc6313`.
14. *YAF - Documentation* [online] [visited on 2020-03-12]. Available from: `https://tools.netsa.cert.org/yaf/yafdpi.html`.
15. MOORE, Andrew W.; ZUEV, Denis; CROGAN, Michael L. Discriminators for Use in Flow-Based Classification. 2005.
16. ESTE, Alice; GRINGOLI, Francesco; SALGARELLI, Luca. On the Stability of the Information Carried by Traffic Flow Features at the Packet Level. *ACM SIGCOMM Computer Communication Review* [online]. 2009, vol. 39, no. 3, pp. 13 [visited on 2020-03-29]. ISSN 01464833. Available from DOI: `10.1145/1568613.1568616`.
17. DAINOTTI, Alberto; PESCAPE, Antonio; SALVO ROSSI, Pierluigi Salvo; IANNELLO, Giulio; PALMIERI, Francesco; VENTRE, Giorgio. QRP07-2: An HMM Approach to Internet Traffic Modeling. In: *IEEE Globecom 2006*. 2006, pp. 1–6. ISSN 1930-529X. Available from DOI: `10.1109/GLOCOM.2006.453`.
18. ANDERSON, Blake; PAUL, Subharthi; MCGREW, David. Deciphering Malware's Use of TLS (without Decryption) [online]. 2016 [visited on 2020-03-01]. Available from arXiv: `1607.01639`.
19. DAINOTTI, Alberto; PESCAPE, Antonio; KIM, Hyun-chul. Traffic Classification through Joint Distributions of Packet-Level Statistics. In: *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*. 2011, pp. 1–6. ISSN 1930-529X. Available from DOI: `10.1109/GLOCOM.2011.6134093`.

20. ANDERSON, Blake; MCGREW, David. Machine Learning for Encrypted Malware Traffic Classification: Accounting for Noisy Labels and Non-Stationarity. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* [online]. Halifax, NS, Canada: Association for Computing Machinery, 2017, pp. 1723–1732 [visited on 2020-03-12]. KDD ’17. ISBN 978-1-4503-4887-4. Available from DOI: `10.1145/3097983.3098163`.

21. ALTHOUSE, John B.; ATKINSON, Jeff; ATKINS, Josh. *Salesforce/Ja3* [online]. 2020 [visited on 2020-03-05]. Available from: `https://github.com/salesforce/ja3`.

22. *SSL Fingerprint JA3* [online] [visited on 2020-03-30]. Available from: `https://ja3er.com/`.

23. *EmpireProject/Empire* [online]. 2020 [visited on 2020-03-30]. Available from: `https://github.com/EmpireProject/Empire`.

24. *SquareLemon* [online] [visited on 2020-03-30]. Available from: `https://blog.squarelemon.com/tls-fingerprinting/`.

25. SADRE, Ramin; SPEROTTO, Anna; PRAS, Aiko. The Effects of DDoS Attacks on Flow Monitoring Applications. In: *2012 IEEE Network Operations and Management Symposium*. 2012, pp. 269–277. ISSN 1542-1201. Available from DOI: `10.1109/NOMS.2012.6211908`.

26. SPEROTTO, Anna; SCHAFFRATH, Gregor; SADRE, Ramin; MORARIU, Cristian; PRAS, Aiko; STILLER, Burkhard. An Overview of IP Flow-Based Intrusion Detection. *IEEE Communications Surveys Tutorials*. 2010, vol. 12, no. 3, pp. 343–356. ISSN 2373-745X. Available from DOI: `10.1109/SURV.2010.032210.00054`.

27. HOFSTEDE, Rick; BARTOŠ, Václav; SPEROTTO, Anna; PRAS, Aiko. Towards Real-Time Intrusion Detection for NetFlow and IPFIX. In: *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. 2013, pp. 227–234. ISSN 2165-963X. Available from DOI: `10.1109/CNSM.2013.6727841`.

28. NEIL, Joshua; UPHOFF, Benjamin; HASH, Curtis; STORLIE, Curtis. Towards Improved Detection of Attackers in Computer Networks: New Edges, Fast Updating, and Host Agents. In: *2013 6th International Symposium on Resilient Control Systems (ISRCS)*. 2013, pp. 218–224. ISSN null. Available from DOI: `10.1109/ISRCS.2013.6623779`.

29. BILGE, Leyla; BALZAROTTI, Davide; ROBERTSON, William; KIRDA, Engin; KRUEGEL, Christopher. Disclosure: Detecting Botnet Command and Control Servers through Large-Scale NetFlow Analysis. In: *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC ’12* [online]. Orlando, Florida: ACM Press, 2012, p. 129 [visited

on 2020-03-04]. ISBN 978-1-4503-1312-4. Available from DOI: 10.1145/2420950.2420969.

30. VELAN, Petr; ČERMÁK, Milan; ČELEDA, Pavel; DRAŠAR, Martin. A Survey of Methods for Encrypted Traffic Classification and Analysis. *International Journal of Network Management* [online]. 2015, vol. 25, no. 5, pp. 355–374 [visited on 2020-03-05]. ISSN 10557148. Available from DOI: 10.1002/nem.1901.

31. *Snort - Network Intrusion Detection & Prevention System* [online] [visited on 2020-03-05]. Available from: https://www.snort.org/.

32. *Suricata* [online] [visited on 2020-03-05]. Available from: https://suricata-ids.org/.

33. ANDERSON, Blake; MCGREW, David. Identifying Encrypted Malware Traffic with Contextual Flow Data. In: *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security - ALSec '16* [online]. Vienna, Austria: ACM Press, 2016, pp. 35–46 [visited on 2020-03-01]. ISBN 978-1-4503-4573-6. Available from DOI: 10.1145/2996758.2996768.

34. MILLER, Brad; HUANG, Ling; JOSEPH, A. D.; TYGAR, J. D. I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis. In: *Privacy Enhancing Technologies* [online]. Springer International Publishing, 2014, vol. 8555, pp. 143–163 [visited on 2020-03-04]. ISBN 978-3-319-08506-7. Available from DOI: 10.1007/978-3-319-08506-7_8.

35. *Fail2ban* [online] [visited on 2020-03-05]. Available from: http://www.fail2ban.org/.

36. JONKER, Mattijs; HOFSTEDE, Rick; SPEROTTO, Anna; PRAS, Aiko. Unveiling Flat Traffic on the Internet: An SSH Attack Case Study. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)* [online]. Ottawa, ON, Canada: IEEE, 2015, pp. 270–278 [visited on 2020-02-11]. ISBN 978-1-4799-8241-7. Available from DOI: 10.1109/INM.2015.7140301.

37. HELLEMONS, Laurens; HENDRIKS, Luuk; HENDRIKS, Luuk; HOFSTEDE, R. J.; SPEROTTO, Anna; SADRE, R.; PRAS, Aiko. SSHCure: A Flow-Based SSH Intrusion Detection System. *Proceedings of the 6th International Conference on Autonomous Infrastructure, Management, and Security (AIMS 2012)* [online]. 2012, pp. 86–97 [visited on 2020-03-05]. Available from DOI: 10.1007/978-3-642-30633-4_11.

38. MCGREW, David; ANDERSON, Blake. *Cisco/Joy* [online]. 2020 [visited on 2020-03-31]. Available from: https://github.com/cisco/joy.

39. *Scikit-Learn: Machine Learning in Python — Scikit-Learn 0.22.2 Documentation* [online] [visited on 2020-03-31]. Available from: https://scikit-learn.org/.

40. RESCORLA, Eric. *The Transport Layer Security (TLS) Protocol Version 1.3* [online] [visited on 2020-04-07]. Available from: `https://tools.ietf.org/html/rfc8446`.

41. RESCORLA, Eric. *The Transport Layer Security (TLS) Protocol Version 1.2* [online] [visited on 2020-04-01]. Available from: `https://tools.ietf.org/html/rfc5246`.

42. *TLS 1.3 Updates the Most Important Security Protocol on the Internet, Delivering Superior Privacy, Security, and Performance* [online] [visited on 2020-04-08]. Available from: `https://ietf.org/blog/tls13/`.

43. *TLS Fingerprinting in the Real World* [online]. 2019 [visited on 2020-04-08]. Available from: `https://blogs.cisco.com/security/tls-fingerprinting-in-the-real-world`.

44. *TLS Fingerprinting with JA3 and JA3S - Salesforce Engineering* [online] [visited on 2020-04-08]. Available from: `https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967`.

45. OKU, Kazuho; WOOD, Christopher; RESCORLA, Eric; SULLIVAN, Nick. *Encrypted Server Name Indication for TLS 1.3* [online] [visited on 2020-04-08]. Available from: `https://tools.ietf.org/html/draft-ietf-tls-esni-06`.

46. ANDREASEN, F. *TLS 1.3 Impact on Network-Based Security* [online]. 2019 [visited on 2020-04-08]. Available from: `https://tools.ietf.org/id/draft-camwinget-tls-use-cases-05.html`.

47. *Usage Statistics and Market Share of Content Management Systems, April 2020* [online] [visited on 2020-04-09]. Available from: `https://w3techs.com/technologies/overview/content_management`.

48. HOFSTEDE, Rick; JONKER, Mattijs; SPEROTTO, Anna; PRAS, Aiko. Flow-Based Web Application Brute-Force Attack and Compromise Detection. *Journal of Network and Systems Management* [online]. 2017, vol. 25, no. 4, pp. 735–758 [visited on 2020-02-11]. ISSN 1064-7570, 1573-7705. ISSN 1064-7570, 1573-7705. Available from DOI: `10.1007/s10922-017-9421-4`.

49. Van der TOORN, Olivier; HOFSTEDE, Rick; JONKER, Mattijs; SPEROTTO, Anna. A First Look at HTTP(S) Intrusion Detection Using NetFlow/IPFIX. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)* [online]. Ottawa, ON, Canada: IEEE, 2015, pp. 862–865 [visited on 2020-02-11]. ISBN 978-1-4799-8241-7. Available from DOI: `10.1109/INM.2015.7140395`.

50. *Nmap/Ncrack* [online]. 2020 [visited on 2020-04-11]. Available from: `https://github.com/nmap/ncrack`.

51. HAUSER, van. *Vanhauser-Thc/Thc-Hydra* [online]. 2020 [visited on 2020-04-11]. Available from: `https://github.com/vanhauser-thc/thc-hydra`.

52. LANJELOT. *Lanjelot/Patator* [online]. 2020 [visited on 2020-04-11]. Available from: `https://github.com/lanjelot/patator`.

53. GARCIA, Sebastian. *Malware Capture Facility Project* [online] [visited on 2020-05-05]. Available from: `https://stratosphereips.org`.

54. CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. SMOTE: Synthetic Minority Over-Sampling Technique. *Journal of Artificial Intelligence Research* [online]. 2002, vol. 16, pp. 321–357 [visited on 2020-05-06]. ISSN 1076-9757. Available from DOI: `10.1613/jair.953`.

55. KE, Guolin; MENG, Qi; FINLEY, Thomas; WANG, Taifeng. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 3149–3157. NIPS'17. ISBN 978-1-5108-6096-4.

56. *CESNET/Nemea* [online]. 2020 [visited on 2020-05-09]. Available from: `https://github.com/CESNET/Nemea`.

57. *CESNET/Ipfixcol2* [online]. 2020 [visited on 2020-05-09]. Available from: `https://github.com/CESNET/ipfixcol2`.

58. *Intrusion Detection Extensible Alert [IDEA]* [online] [visited on 2020-05-09]. Available from: `https://idea.cesnet.cz/`.

59. BREIMAN, L.; FRIEDMAN, J.; OLSHEN, R.; STONE, C. Classification and Regression Trees. 1984.

60. HASTIE, Trevor; ROSSET, Saharon; ZHU, Ji; ZOU, Hui. Multi-Class AdaBoost. *Statistics and Its Interface* [online]. 2009, vol. 2, no. 3, pp. 349–360 [visited on 2020-05-06]. ISSN 19387989, 19387997. ISSN 19387989, 19387997. Available from DOI: `10.4310/SII.2009.v2.n3.a8`.

61. FORMAN, George; SCHOLZ, Martin. Apples-to-Apples in Cross-Validation Studies: Pitfalls in Classifier Performance Measurement. Vol. 12, no. 1, pp. 9.

Appendix A

# Acronyms

| | |
|---|---|
| **AJAX** | Asynchronous JavaScript and XML |
| **CMS** | Content management system |
| **DNS** | Domain Name System |
| **DPI** | Deep packet inspection |
| **HTML** | Hypertext Markup Language |
| **HTTP** | Hypertext Transfer Protocol |
| **HTTPS** | Hypertext Transfer Protocol Secure |
| **IDEA** | Intrusion Detection Extensible Alert |
| **IE** | Information Element |
| **IPFIX** | Internet Protocol Flow Information Export |
| **ISP** | Internet service provider |
| **JSON** | JavaScript Object Notation |
| **MPLS** | Multiprotocol Label Switching |
| **NAT** | Network address translation |
| **PPI** | Per-Packet information |
| **RFC** | Request for Comments |
| **TCP** | Transmission Control Protocol |
| **TLS** | Transport Layer Security |
| **UDP** | User Datagram Protocol |

Appendix B

# Contents of enclosed USB drive

```
readme.txt ........ the file with USB drive contents description
src ............... the directory of source codes
  detection ....... the directory of source codes of the detection module
  thesis .......... the directory of LaTeX source codes of the thesis
text .............. the thesis text directory
  thesis.pdf ...... the thesis text in the PDF format
```