FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Learning from HTTP/2 encrypted traffic: a machine learning-based analysis tool

Diogo Belarmino Coelho Marques

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ricardo Santos Morla

July 25, 2018

Learning from HTTP/2 encrypted traffic: a machine learning-based analysis tool

Diogo Belarmino Coelho Marques

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Miguel Pimenta Monteiro, PhD

External Examiner: Pedro Brandão, PhD

Supervisor: Ricardo Morla, PhD

July 25, 2018

Abstract

Secure protocols in computer networks do not prevent malicious third-parties from uncovering potentially sensitive information. Side-channel vulnerabilities allow passive attackers to eavesdrop on encrypted network traffic just by observing normal system behavior, retrieving relevant metadata ("fingerprints") without actively interacting with the system or decrypting such information.

The goal is to study the impact of web browsers and web engines on the ability to perform website fingerprinting over encrypted HTTP/2 connections. These attacks may facilitate third-parties not only to identify the structure of the subject website, but also understanding user interaction patterns (for instance, what pages were visited and what operations were performed during the user session), thus compromising user anonymity and privacy.

The solution consists of developing a capture and analysis tool for encrypted and unencrypted HTTP/2 web traffic. Both the capture process and web browser interaction are automated to ensure minimal user interaction and replication of this procedure on multiple machines. For testing the implementation, a website consisting of several pages with different content was developed, targeting multiple web engines that implement the aforementioned protocol, which were then deployed on isolated testing environments using virtualization.

Traffic flows between browser and server are captured within this controlled environment and then manually analyzed using *Wireshark* and the visualization component. The purpose is to identify unique encrypted traffic characteristics that could potentially model machine learning classification problems, such as inferring about the count and type of HTTP/2 frames obfuscated within TLS records. The parsing component extracts these features from packet captures, which are then fed into the machine learning component. Different datasets are generated in order to compare protocol implementation differences. Results have shown a perceivable decrease in prediction accuracy for classifiers trained for a specific browser or engine when classifying samples from their counterparts. Frame types could be inferred with reasonable accuracy among all experiments, while frame count only yielded good results for TLS records with up to two frames.

Resumo

Os protocolos seguros nas redes de computadores não são suficientes para impedir que terceiros mal intencionados revelem informações potencialmente sensíveis. As vulnerabilidades *side-channel* permitem a atacantes passivos escutar tráfego encriptado na rede pela observação do comportamento normal do sistema, obtendo metadados relevantes ("impressões digitais" ou *fingerprints*) sem interagir de forma ativa com o sistema nem descodificando essa informação.

O objetivo é estudar o impacto dos *web browsers* e *web engines* na possibilidade de realizar *fingerprinting* em *websites* implementados sobre ligações HTTP/2 encriptadas. Estes ataques não só potenciam terceiros a identificar a estrutura do *website* em questão, como também a perceber os padrões de interação dos utilizadores (por exemplo, que páginas visitou e que operações realizou durante a sessão), comprometendo assim o anonimato e a privacidade dos utilizadores.

A solução consiste em desenvolver uma ferramenta para captura e análise de tráfego web HTTP/2 encriptado e não encriptado. Tanto o processo de captura como a interação com os *web browsers* são automatizados para garantir a mínima interação do utilizador e replicação deste procedimento em máquinas diferentes. Com vista a testar a implementação foi desenvolvido um *website* constituído por várias páginas com conteúdos diferentes, recorrendo a várias *web engines* que implementam este protocolo, que então foram instalados em ambientes de teste isolados baseados em máquinas virtuais.

Os fluxos de tráfego entre cliente e servidor são capturados neste ambiente controlado e analisados manualmente recorrendo ao *Wireshark* e à componente de visualização com vista a identificar características do tráfego encriptado com potencialidades para modelar problemas de classificação em *machine learning*, nomeadamente inferir quanto ao tipo e número de tramas HTTP/2 presentes num dado registo TLS. A componente de *parsing* extrai estas características das capturas, que servem como entrada à componente de *machine learning*. Foram gerados diferentes *datasets* de forma a comparar diferenças na implementação deste protocolo. Os resultados mostraram uma diminuição percetível na precisão para os classificadores treinados para um *browser* ou *engine* específico ao classificarem amostras dos seus homólogos. Os tipos de tramas foram inferidos com precisão razoável na globalidade das experiências realizadas, enquanto o número de tramas apresentou bons resultados apenas para registos TLS com até duas tramas.

“If you think technology can solve your security problems, then you don’t understand the problems, and you don’t understand the technology.”

Bruce Schneier

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Goals . . . 3
- 1.4 Document structure . . . 3

2 Literature review **5**
- 2.1 Technical background . . . 5
  - 2.1.1 HTTP/2 . . . 5
  - 2.1.2 Support vector machines . . . 6
  - 2.1.3 Web engines . . . 7
- 2.2 Related work . . . 8
- 2.3 Discussion . . . 9

3 Solution **13**
- 3.1 Methodology . . . 13
- 3.2 Architecture . . . 14
  - 3.2.1 Capture automation tool . . . 14
  - 3.2.2 Analysis tool . . . 17
- 3.3 Parser component . . . 17

4 Visualization component **21**
- 4.1 Implementation . . . 21
- 4.2 Results . . . 23
  - 4.2.1 Web objects sequence . . . 24
  - 4.2.2 Stream dependency graph . . . 25
  - 4.2.3 Length distribution . . . 25
- 4.3 Discussion . . . 32

5 Machine learning component **33**
- 5.1 Implementation . . . 34
- 5.2 Results . . . 35
  - 5.2.1 Web Engine - Apache . . . 36
  - 5.2.2 Web Engine - NGINX 1.14.0 . . . 38
  - 5.2.3 Web Engine - NGINX 1.10.3 . . . 40
  - 5.2.4 Web Browser - Google Chrome . . . 42
  - 5.2.5 Web Browser - Mozilla Firefox . . . 43
- 5.3 Discussion . . . 43

6 Conclusions **45**
6.1 Future work 46

References **47**

List of Figures

2.1 Market share of the top five web engines on the global market for active websites, according to the *Netcraft* survey, January 2018. 7

2.2 Market share of the top five web engines on the busiest 1 million websites according to the *Netcraft* survey, January 2018. 8

2.3 Chart showing the percentages of websites using various web engines broken down by ranking, according to the *W3Techs* survey, February 2018. 11

3.1 Architecture diagram for the capture automation tool 14

3.2 Architecture diagram for the HTTP/2 traffic analysis tool 17

3.3 Example tree of the capture directory after running both *fingerprint-preprocess.py* and *fingerprint-analysis.py* scripts. 20

4.1 Sequence of events for web objects requested by the browser 24

4.2 Stream dependency graph 25

4.3 Length distribution of frames (absolute and relative units, box-and-whisker plot) . 27

4.4 Length distribution of frames sent by the client (absolute and relative units, box-and-whisker plot) 28

4.5 Length distribution of frames sent by the server (absolute and relative units, box-and-whisker plot) 29

4.6 Length distribution of the first frames from each stream (absolute and relative units, box-and-whisker plot) 30

4.7 Length distribution of the last frames from each stream (absolute and relative units, box-and-whisker plot) 31

5.1 Confusion matrices for the “engine-apache” experiment, “ehes” classifier: cross-validation results (Apache *versus* Apache), prediction results using samples from “engine-nginx-latest” (Apache *versus* NGINX 1.14.0), prediction results using samples from “engine-nginx-stable” (Apache *versus* NGINX 1.10.3). 36

5.2 Confusion matrices for the “engine-apache” experiment, “frame_count” classifier: cross-validation results (Apache *versus* Apache), prediction results using samples from “engine-nginx-latest” (Apache *versus* NGINX 1.14.0), prediction results using samples from “engine-nginx-stable” (Apache *versus* NGINX 1.10.3). 36

5.3 Confusion matrices for the “engine-apache” experiment, “frame_types” classifier: cross-validation results (Apache *versus* Apache), prediction results using samples from “engine-nginx-latest” (Apache *versus* NGINX 1.14.0), prediction results using samples from “engine-nginx-stable” (Apache *versus* NGINX 1.10.3). 37

5.4 Confusion matrices for the "engine-nginx-latest" experiment, "ehes" classifier: cross-validation results (NGINX 1.14.0 *versus* NGINX 1.14.0), prediction results using samples from "engine-apache" (NGINX 1.14.0 *versus* Apache), prediction results using samples from "engine-nginx-stable" (NGINX 1.14.0 *versus* NGINX 1.10.3). 39
5.5 Confusion matrices for the "engine-nginx-latest" experiment, "frame_count" classifier: cross-validation results (NGINX 1.14.0 *versus* NGINX 1.14.0), prediction results using samples from "engine-apache" (NGINX 1.14.0 *versus* Apache), prediction results using samples from "engine-nginx-stable" (NGINX 1.14.0 *versus* NGINX 1.10.3). 39
5.6 Confusion matrices for the "engine-nginx-latest" experiment, "frame_types" classifier: cross-validation results (NGINX 1.14.0 *versus* NGINX 1.14.0), prediction results using samples from "engine-apache" (NGINX 1.14.0 *versus* Apache), prediction results using samples from "engine-nginx-stable" (NGINX 1.14.0 *versus* NGINX 1.10.3). 39
5.7 Confusion matrices for the "engine-nginx-stable" experiment, "ehes" classifier: cross-validation results (NGINX 1.10.3 *versus* NGINX 1.10.3), prediction results using samples from "engine-apache" (NGINX 1.10.3 *versus* Apache), prediction results using samples from "engine-nginx-latest" (NGINX 1.10.3 *versus* NGINX 1.14.0). 40
5.8 Confusion matrices for the "engine-nginx-stable" experiment, "frame_count" classifier: cross-validation results (NGINX 1.10.3 *versus* NGINX 1.10.3), prediction results using samples from "engine-apache" (NGINX 1.10.3 *versus* Apache), prediction results using samples from "engine-nginx-latest" (NGINX 1.10.3 *versus* NGINX 1.14.0). 40
5.9 Confusion matrices for the "engine-nginx-stable" experiment, "frame_types" classifier: cross-validation results (NGINX 1.10.3 *versus* NGINX 1.10.3), prediction results using samples from "engine-apache" (NGINX 1.10.3 *versus* Apache), prediction results using samples from "engine-nginx-latest" (NGINX 1.10.3 *versus* NGINX 1.14.0). 41
5.10 Confusion matrices for the "browser-chrome" experiment: cross-validation results for "ehes", "frame_count" and "frame_type" classifiers 42
5.11 Confusion matrices for the "browser-chrome" experiment: prediction results for the "ehes", "frame_count" and "frame_type" classifiers using samples from "browser-firefox". 42
5.12 Confusion matrices for the "browser-firefox" experiment: cross-validation results for "ehes", "frame_count" and "frame_type" classifiers 43
5.13 Confusion matrices for the "browser-firefox" experiment: prediction results for the "ehes", frame_count and "frame_type" classifiers using samples from "browser-chrome". 43

List of Tables

5.1 Cross-validation accuracy metrics for five distinct machine learning algorithms (best accuracy values for each classifier in bold) using samples from "sample-0". 35

5.2 Number of correctly and incorrectly classified samples and classification accuracy for the cross validation results using samples from "sample-25" and "sample-50". 35

5.3 Number of correctly and incorrectly classified samples and classification accuracy for the cross validation results using samples from "sample-100" and "sample-0". 35

5.4 Number of correctly and incorrectly classified samples and classification accuracy for cross-validation using samples from "engine-apache". 37

5.5 Number of correctly and incorrectly classified samples and prediction accuracy using samples from "engine-nginx-latest" and "engine-nginx-stable"; the three classifiers were trained using samples from *engine-apache*. 37

5.6 Number of correctly and incorrectly classified samples and classification accuracy for cross-validation using samples from the "engine-nginx-stable" experiment. . . 38

5.7 Number of correctly and incorrectly classified samples and prediction accuracy using samples from "engine-apache" and "engine-nginx-stable"; the three classifiers were trained using samples from *engine-nginx-latest*. 38

5.8 Number of correctly and incorrectly classified samples and classification accuracy for cross-validation using samples from "engine-nginx-stable". 41

5.9 Number of correctly and incorrectly classified samples and prediction accuracy using samples from "engine-apache" and "engine-nginx-latest"; the classifiers were trained using samples from *engine-nginx-stable*. 41

5.10 Number of correctly and incorrectly classified samples and classification accuracy from the cross-validation results and prediction results using samples from "browser-firefox". 42

5.11 Number of correctly and incorrectly classified samples and classification accuracy from the cross-validation results and prediction results using samples from "browser-chrome". 43

LIST OF TABLES

Abbreviations

CART	Classification and Regression Trees
CPU	Central Processing Unit
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol *(Secure)*
IETF	Internet Engineering Task Force
ISP	Internet Service Provider
IP	Internet Protocol
JSON	JavaScript Object Notation
QUIC	Quick UDP Internet Connections
SVM	Support Vector Machine
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
VoIP	Voice over Internet Protocol
VPN	Virtual Private Network
YAML	Yet Another Markup Language

Chapter 1

Introduction

Web engines and web application frameworks can have an effect on the fingerprint that users leave on the Internet when visiting websites. Understanding which web browser and web engine is responsible for a given flow of encrypted data may help better identify websites and their usage. This dissertation evaluates the impact of different web browsers and web engines on website fingerprinting.

1.1 Context

Secure web protocols encrypt traffic so that only the client web browser and the web server hosting that website should be the only parties able to access the contents of this communication [Fel10]. Despite being encrypted, the communication channel established between the client and the server does not prevent specific types of attacks performed by third-parties that uncover potentially sensitive information without accessing or taking possession of the unencrypted data.

Side-channel vulnerabilities allow third-parties to infer such information just by observing normal behavior of a software system. The attacker is a passive observer, because it does not actively interact with the system. Side-channel leaks are discussed broadly in many contexts, not necessarily about encrypted communications. Information can be leaked through anything [CWWZ10], including electromagnetic signals, shared memory, registers and files between processes and CPU usage metrics. This is why side-channel attacks have been historically used to defeat cryptography systems, but also apply to encrypted network connections, such as HTTPS and VPN, which will be explored in the next chapter.

In the case of network connections, an attacker may eavesdrop on the encrypted traffic generated on the network side and infer some of its properties through metadata and packet flow information, without performing deep payload inspection (accessing the actual content of the objects or messages being exchanged between both parties in the encrypted connection). The attacker does

not have means of decrypting the information and thus does not understand the requests sent by the user browser, nor the responses and reply messages sent by the server.

Such attacks could enable a third-party to learn about user interaction on a given website without accessing the actual information being exchanged between client and server. This third-party could just be an individual with a malicious intent, our own internet service provider (ISP) or even a government agency [DDPH17] with the ultimate goal of collecting open-source intelligence from users or a particular target of investigation.

Felten [Fel10] gives us the following example of a side-channel attack against websites and users: consider a search engine that auto-completes search queries: when the user starts typing a query, the search engine gives a list of suggested queries that start with whatever characters were typed so far. When the user types the first letter on the search query, the search engine page will send that character to the server, and the server will send back a list of suggested completions. Unfortunately, the size of that suggested completion list will depend on which character the user typed, so an eavesdropper can use the size of the encrypted response to deduce which letter was typed. When the user types the second letter of their search query, another request will be sent to the server, and another encrypted reply will be returned to the user, which will again have a distinct size, allowing the attacker (who already knows about the first character) to deduce the second character, and so the following characters. In the end, the eavesdropper will know exactly which search query was typed by the user.

1.2 Motivation

The motivation for this dissertation comes primarily from the fact that web fingerprinting might pose an even greater risk on the privacy and anonymity of Internet users, although internet service providers and the enterprise sometimes have the best interests when performing fingerprinting on their users. For instance, system administrators might want to prevent access to forbidden websites or provision the quality of service on a specific website, but generally fingerprinting is performed by a third-party with some malicious intent, mostly an intruder.

Another relevant factor is the lack of attention given to enforcing programming standards and adequate choice of web technologies as potential countermeasures to fingerprinting. Previous research efforts conducted on network privacy and traffic analysis in the last two decades focused on studying the viability of identifying and distinguishing between multiple protocols and encrypted connections (including BitTorrent, VPN and Tor networks), and thus only evaluated the amount of features and metadata that could be extracted from the network packets, rather than findings ways of preventing such analysis by third-parties.

Lastly, the innovative aspect of this dissertation is the exclusive usage of the HTTP/2 [BPT15] protocol and technologies. There is very little information regarding fingerprinting performed on this new revision of the well-known web protocol. The web applications developed for the experiments will be designed with HTTP/2 compatibility in mind.

1.3 Goals

The main goal of this project is to study the impact of web browsers and web engines on the characteristics and features of encrypted HTTP/2 traffic generated by websites during a typical user interaction. Additionally, assess whether some implementations are more or less vulnerable to fingerprinting than others due to relaxed/stricter security and programming standards, and evaluate what sensitive information can be inferred with great confidence from encrypted web traffic. In order to support the primary objective and develop a solid solution for the problem, a secondary goal was set, consisting of the development of a capture automation and analysis tool. The tool must be as generic as possible, allowing it to be extended beyond the purposes of this project.

1.4 Document structure

Besides this general overview and description of the dissertation objectives, this report compromises of the following chapters:

- Chapter 2 describes the technical background on the subject of HTTP/2 and machine learning, and presents previous research efforts on side-channel vulnerabilities and encrypted network traffic classification.
- Chapter 3 describes the solution approach and methodology, the implementation details on the capture automation and analysis tools and presents an architecture diagram of the software components and their interactions.
- Chapter 4 describes the visualization component of the analysis tool, the implementation details, presents results and discusses the usefulness of this component for the machine learning approach.
- Chapter 5 describes the machine learning component of the analysis tool, the implementation details, characterizes the features, classifiers and datasets used, presents results and discusses the differences.
- Chapter 6 draws some conclusions regarding the relevance of the problem, the research importance of the results and suggests possible future work.

Chapter 2

Literature review

This chapter describes the technical background on the subject of HTTP/2 and machine learning, and then proceeds to present previous research efforts on side-channel vulnerabilities and encrypted network traffic classification.

2.1 Technical background

This section describes the technical background on the HTTP/2 protocol specification, machine learning algorithms, existing web browsers and web engines, and the state of their HTTP/2 implementation.

2.1.1 HTTP/2

HTTP/2 is a major revision of the traditional HTTP/1.1 protocol widely used on websites, and a relatively recent standard, proposed on RFC 7540 [BPT15] (May 2015) and since then has been gradually adopted by the biggest players in the market. This standard brings performance and security related-features such as synchronous communication between client and server: clients no longer need to wait for the server response to the current request to issue the next request (pipelining), and servers no longer need to wait until the response is sent to begin sending the next response (multiplexing). Multiplexing and pipelining techniques should provide performance improvements, even on bandwidth-starved or high latency connections.

HTTP/2 enables more efficient processing of messages through binary message framing and provides an optimized transport for HTTP semantics. The protocol supports all HTTP/1.1 core features while attempting to be more efficient in several ways. The smallest unit of communication within an HTTP/2 connection is called a frame, which may assume distinct types that serve different communication purposes. A frame consists of an header and a variable-length sequence of bytes structured according to the frame type.

A stream is an independent, bidirectional sequence of frames exchanged between the client and server within an HTTP/2 connection. A single HTTP/2 connection may contain multiple concurrently open streams, with either endpoint interleaving frames from multiple streams. Streams can be established and used unilaterally or shared by either the client or the server, and they can be closed by either endpoint. Streams are identified by an unsigned 31-bit integer, which is assigned by the endpoint initiating the stream. Streams initiated by a client have odd-numbered identifiers, while those initiated by the server have even-numbered stream identifiers. There are two exceptions: streams associated with connection control messages have an identifier of 0, while HTTP/1.1 requests that are upgraded to HTTP/2 assume an identifier of 1. Since these two identifiers are reserved, they cannot be used to establish a new stream.

Mozilla Firefox was the first browser supporting the HTTP/2 final draft specification, which was enabled by default starting from version 36 [Fou15], released in February 2015. Google announced complete HTTP/2 support in Chrome since version 41 [Ven15], which was released in March 2015.

2.1.2 Support vector machines

First proposed by Vapnik [CV95], support vector machines are a binary supervised classification algorithm which transforms a non-linear classification problem into a linear problem. Support vector machines interpret the training samples as points on a multi-dimensional vector space, whose coordinates are the components of the feature vector. The objective is to find a set of surfaces partitioning this space and perfectly separating points belonging to different classes. However, points might be spread out in the space if the problem is non-linear, and describing such extremely complex surfaces may be difficult, when not impossible, to find in a reasonable time.

The fundamental idea of support vector machines is then to map, by the means of a kernel function, the training points in a newly transformed space, usually with higher or even infinite dimensions, where points can be separated by the less complex surface possible, an hyper-plane. In the target space, support vector machines must basically solve the optimization problem of finding the hyper-plane, which separates points belonging to different classes and has the maximum distance from points of either class. The training samples that fall on the margin and identify the hyper-plane are called support vectors. By the end of the training phase, support vector machines produce a model composed of the kernel function parameters and a collection of support vectors describing the partitioning of the target space. During the classification phase, the algorithm classifies new points according to the portion of space they belong to, making classification much less computationally expensive than training.

Since support vector machines are predominantly binary classifiers, some workarounds and adaptation are required for them to properly function with multi-class classification problems. The strategy often adopted is the one-versus-one, where a model for each pair of classes is built and the classification decision is based on a majority voting of all binary models.

Support vector machines were shown to be one of the most effective machine learning algorithms for supervised learning applications, yielding good performance out-of-the box without

much adjustments, especially in complex feature spaces, and have showed particularly good performance in the field of traffic classification. Several kernel functions are available, but usually opting for a Gaussian kernel exhibits the best accuracy. The most notorious drawback of support vector machines is that models in the multidimensional space cannot be interpreted by humans and there is no way to really understand the reason why a model is considered good or bad. Another but equally important drawback is that the classification process may require a fair amount of computation. The number of operations performed is proportional to the number of support vectors (the representative samples) per each class. When the number of classes is large (in the order of hundreds or thousands applications), the computational cost can be prohibitive.

2.1.3 Web engines

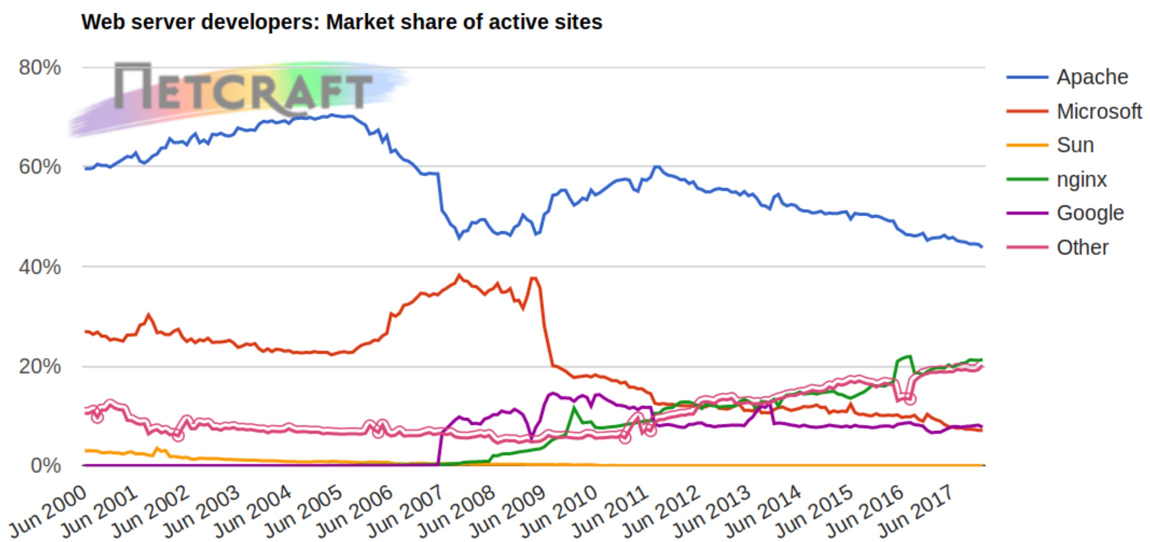

Figure 2.1: Market share of the top five web engines on the global market for active websites, according to the *Netcraft* survey, January 2018.

According to a recent survey from January 2018, *Apache* remains the leading web engine across the global market with a 38.2% share of Internet domains registered [Net18], despite the current trend showing that *Apache* servers could be overtaken by *NGINX* in the upcoming years, in a similar fashion to what happened to *Microsoft IIS* recently. *NGINX* has continued to steadily increase domain share and popularity among websites in the past ten years, while *Apache* has been experiencing a general decline of market share in them same time period.

The persistent and steady growth of *NGINX* was also noticed in every other metric of the *Netcraft* survey conducted in January 2018, gaining the largest number of websites, active websites (Figure 2.1), as well as increasing its presence among the top million websites (Figure 2.2). *NGINX* is now being used by 23.5% of web-facing computer servers and 30.5% of the top million websites, but *Apache* still has the largest number of active websites, computers, domains and top-million websites across the globe. The only metric in which *Apache* did not take the lead was

registered host names, where Microsoft has a total of 575 million websites. *Microsoft IIS* had the second largest number of domains in the survey, but ranked third overall after being overtaken by *NGINX* in October 2017.

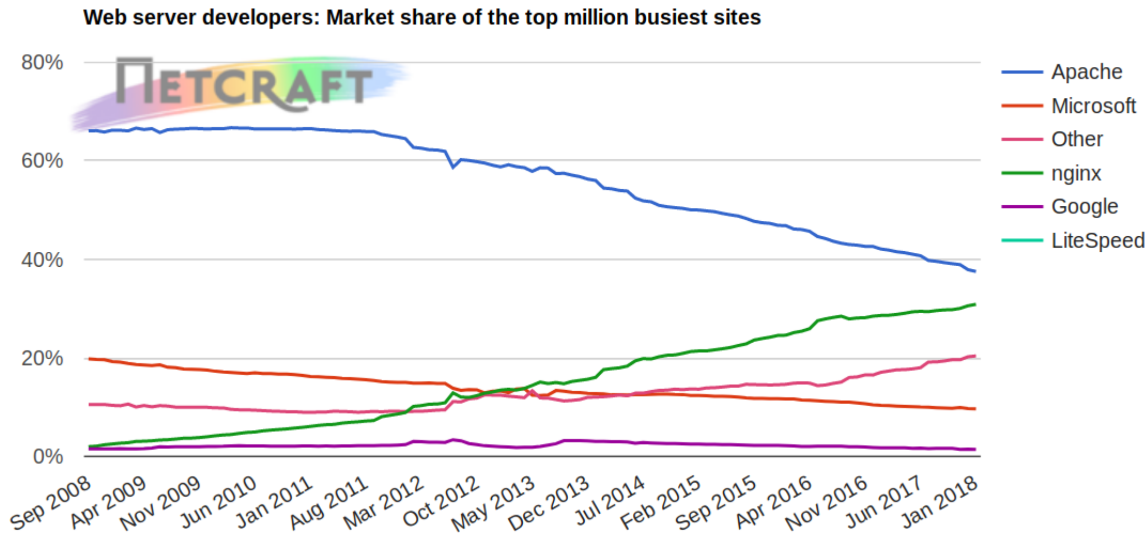

Figure 2.2: Market share of the top five web engines on the busiest 1 million websites according to the *Netcraft* survey, January 2018.

Results from the *W3Techs* [W3T18] survey for February 2018 (Figure 2.3) tell a similar story, with *Apache* being used in 47.5% of the websites on their list, and *NGINX* shortly following it taking 36.7% of the market share. Focusing only on the top million busiest websites, we observe that this segment is dominated by *NGINX*, reaching 44.2% market share, while *Apache* is placed second with 39.0% of the top million busiest websites running on this web engine. *Microsoft IIS* market share values do not look very favorable in either of these surveys, with the market being dominated by open-source technologies. *Google Servers* show an interesting result when we consider only the top 1,000 websites, overtaking *Microsoft IIS*.

Before interpreting surveys and graphs from *W3Techs*, take note of some assumptions made by the authors of this website. *W3Techs* investigates technologies of websites, not individual web pages. Moreover, only the top 10 million websites are included in the statistics in order to limit the impact of spam domains. Their source of website popularity rankings is provided by *Alexa* and sub-domains are not considered to be separate websites. For instance, `sub1.example.com` and `sub2.example.com` are considered to belong to the same website as `example.com`. This means that the sub-domains of `blogger.com`, `wordpress.com` and similar sites count as a single website.

2.2 Related work

This section presents several related works and research efforts on side-channel attacks over encrypted communication channels for network traffic classification.

The *Tor* network [Pro18] is a group of volunteer-operated servers that allows people to improve their privacy and security on the Internet. *Tor* users employ this network by connecting through a series of virtual tunnels rather than making a direct connection, thus allowing both organizations and individuals to share information over public networks without compromising their privacy. However, numerous side-channel attacks on the Tor network have been reported [SRBS17], focusing on ingress-egress traffic correlation, the specifics of the onioning protocol and active attacks from compromised browsers or *Tor* nodes. The goal for these attacks was to infer the type of application being accessed or identifying the user. These attacks were motivated from previous attempts and extensive research performed on more traditional VPN tunnels with the aim of identifying web pages.

Encrypted voice and video traffic using voice and video-specific protocols have also been subject to side-channel attacks with the intent of uncovering spoken sentences in encrypted *VoIP* (voice over Internet protocol) communications using applications such as *Skype*. Previous work in this subject has shown that combining variable bit rate compression codecs with length-preserving encryption leaks information about *VoIP* conversations. Charles Wright et al., in the article called "Spot me if you can — uncovering spoken phrases in encrypted *VoIP* conversations" [WBC+08] showed that a Markov model trained using speaker-independent and phrase-independent data could detect the presence of some phrases within encrypted *VoIP* calls with recall and precision exceeding 90%. The authors evaluated the impact of noise, dictionary size and word variation. The results showed that an eavesdropper could easily identify a variety of phrases in a number of realistic settings. The default encryption transforms of the voice protocol used do not specify any sort of padding. Although padding could introduce inefficiencies into real-time protocols, the authors conclude that padding offers significant confidentiality benefits for *VoIP* calls.

Regarding HTTP/2 side-channel vulnerabilities, Morla [Mor17] evaluated the current state of multiplexing and pipelining implementation on popular websites and their impact in preventing HTTP response sizes estimation from encrypted TLS records. The paper states that the current extent of these techniques is limited on most websites to actually provide security benefits over the HTTP/1.1 protocol, nevertheless attacking pipelining is still viable, although there's a perceivable increase in object size estimation error.

2.3 Discussion

Both encrypted traffic analysis and fingerprinting have proved to be successful, regardless of the network protocol being analyzed, ranging from *YouTube* video title classification [DDPH17] from encrypted video streams to user and website identification on virtual private networks and the Tor anonymous network. With the notorious amount of machine learning algorithms and techniques currently available and ready to be used, further research and experimentation beyond this report is required in order to find which algorithms are best suited and most accurate for solving this problem. Authors have suggested different algorithms for different classification purposes and different protocols, and have mostly succeeded, obtaining classification accuracy up to 90%.

There are few HTTP/2 analysis tools available in the public domain, such as *h2i*[1] and *nghttp2*[2], although they act as HTTP/2 clients and perform live traffic capture, sending requests and receiving responses to/from the server, and then displaying frame information and statistics to the terminal output. These tools do not support raw packet captures, such as those in the PCAP format generated by *tcpdump*. There were previous efforts in web engine fingerprinting, such as identifying the name and version being used. Although this task seems trivial, given that the server identifies itself in the response headers, there are known spoofing techniques that most web engines and network administrators employ to prevent attacks targeted at that specific engine. SSL Labs [Lab09] have performed HTTP/1.1 client fingerprinting by analyzing unencrypted information from TLS handshakes, such as the specific combination of cipher suites advertised by the web browser. Since HTTP/2 is a relatively new standard, most research conducted in the past targeted HTTP/1.1 protocol with TLS encryption, and thus there are no known related works for the newer revision of this protocol that specifically deal with fingerprinting using encrypted application data analysis.

[1] https://github.com/golang/net/tree/master/http2/h2i
[2] https://github.com/nghttp2/nghttp2

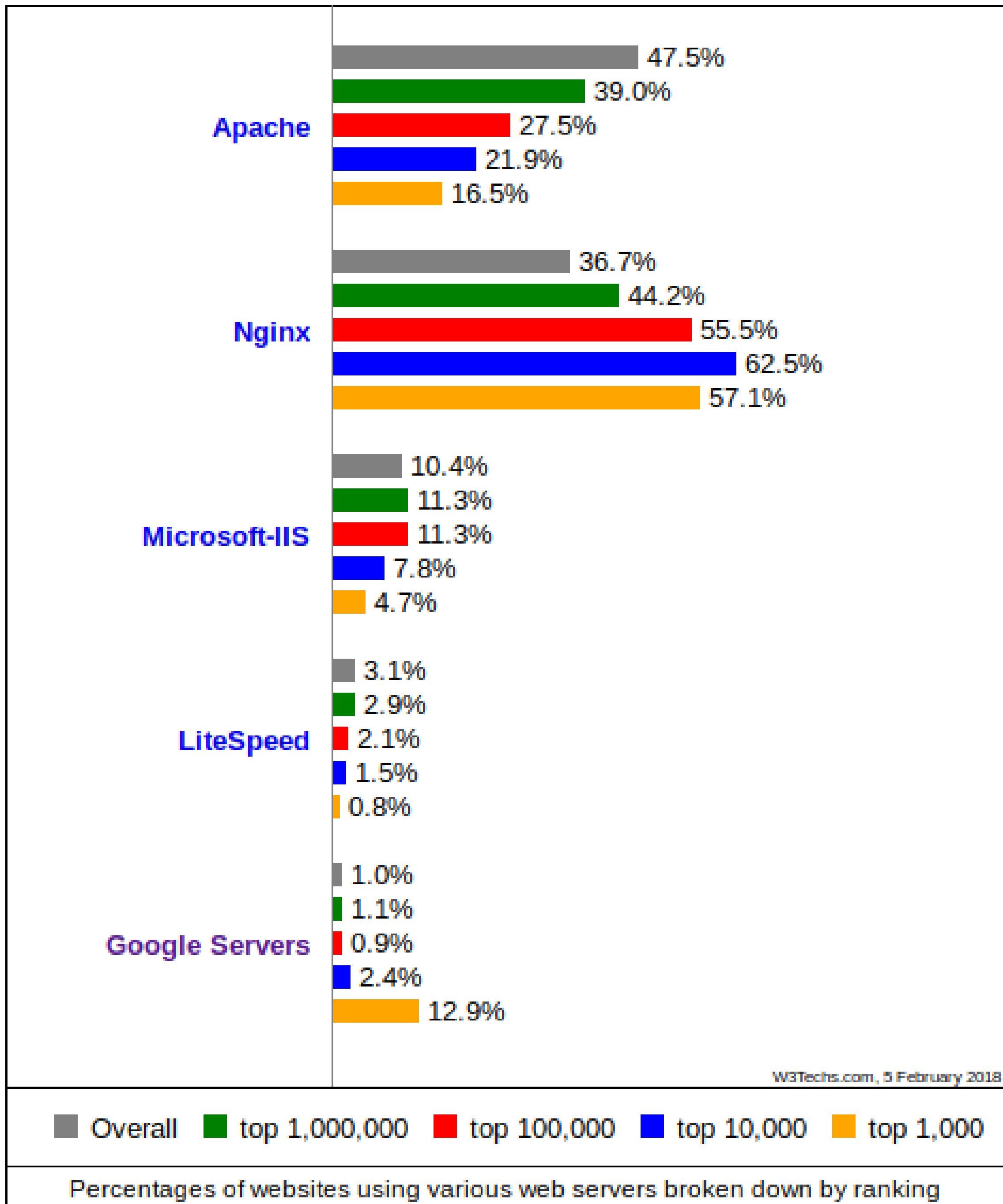

Figure 2.3: Chart showing the percentages of websites using various web engines broken down by ranking, according to the *W3Techs* survey, February 2018.

Chapter 3

Solution

This chapter describes the solution approach and methodology, the implementation details on the capture automation and analysis tools, and proceeds to present an architecture diagram of the software components and their interactions.

3.1 Methodology

The proposed solution consists of the following implementation and testing steps:

1. Conceptualize a website with several pages consisting of text and multimedia content (e.g. audio clips, images or streaming video). Having multiple pages request different content should help with the task of identifying distinct traffic features and content types. Implement the website using combinations of different web engines.

2. Automate the deployment of the website within a controlled environment using Docker as virtualization software. Docker employs lightweight virtual machines (containers) powered by industry-standard and open-source technology from *VirtualBox*.

3. Manually analyze the generated traffic using *Wireshark* and charts generated by the visualization component in order to find and criteria and patterns for inferring information about HTTP/2 frames from encrypted traffic. Packet order, application data (record length and type), packet direction and packet length are some examples of metrics and meta-information provided by the encrypted traffic. Traffic should be preprocessed in order to reduce "background noise", that is, traffic flows that were generated during the experiments but are irrelevant to what is being analyzed and thus does not provide meaningful information. This step is also responsible for translating traffic metadata and the presence or absence of a given feature to a more adequate representation that can be provided as input for the machine learning algorithms.

4. Implement several machine learning algorithms for estimating the count and types of HTTP/2 frames obfuscated within TLS records. This step should not reinvent the wheel, as there are

currently many capable machine learning algorithms for encrypted traffic classification providing the desired level of accuracy.

3.2 Architecture

There are two baseline components that were developed as part of the solution to this problem: the capture automation tool (supported by virtual machines running different web engines) and the analysis tool, further divided into three subcomponents: parser, machine learning and visualization.

3.2.1 Capture automation tool

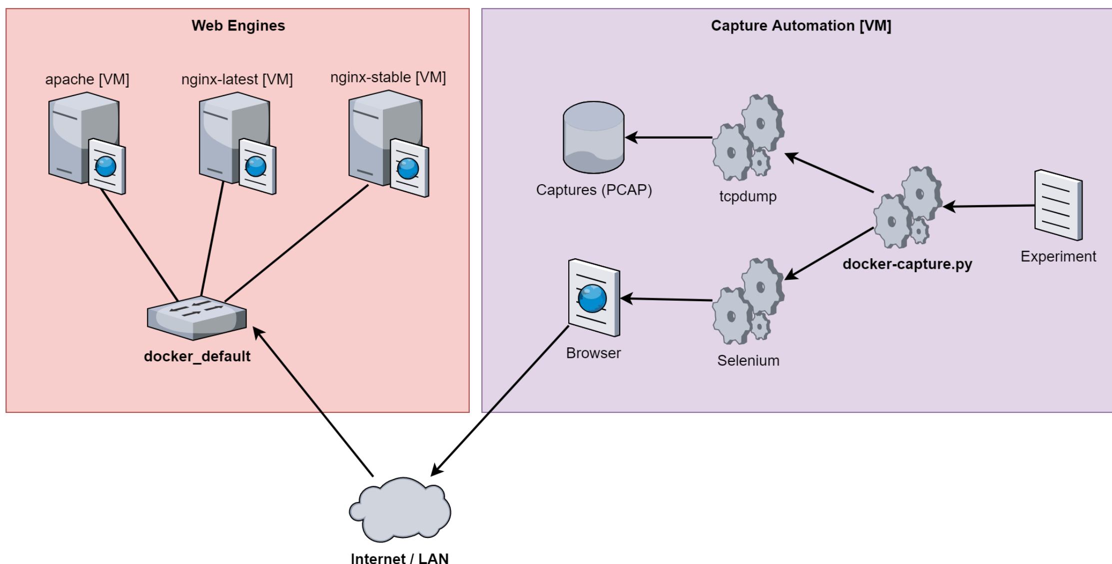

Figure 3.1: Architecture diagram for the capture automation tool

In order to automate the process of generating actual data to feed the visualization and machine learning components, the website capture automation tool was developed. Figure 3.1 presents the architecture of this tool, which implements a minimal and human-readable scripting language based on YAML (Yet Another Markup Language) to define website capture experiments. The automation tool was designed to run inside a virtual machine or a Docker container, since virtual network interfaces are isolated from the host (a local machine running a single or multiple virtual machines). This tool is a simple application written in the Python programming language that reads the user-defined experiment script or a directory containing multiple script, initializes a *sandboxed* environment for running experiments on web browsers, visits the websites specified inside the experiment script, and writes both inbound and outbound traffic for every combination of browser, website and iteration to individual files inside the virtual machine file system. Each experiment script must specify the following parameters:

```yaml
id: gallery-1
repeat: 5
browsers:
  - chrome
  - firefox
wondershaper:
  enable: no
  upload: 1024
  download: 8192
websites:
  - id: apache
    url: https://192.168.109.149:4433/gallery-1.php
  - id: nginx-latest
    url: https://192.168.109.149:4431/gallery-1.php
  - id: nginx-stable
    url: https://192.168.109.149:4432/gallery-1.php
```

Listing 1: Example experiment script (gallery-1.yml)

- **id**: experiment name (must be a unique identifier);
- **repeat**: repetition count (the amount of times this experiment should be repeated for every combination of browsers and websites);
- **browsers**: list of web browsers where the experiment should be performed
- **wondershaper**: *Wondershaper* configuration parameters; whether to enable or disable network traffic shaping ("enable"), the egress bandwidth ("upload") and ingress bandwidth limits ("download") for the default network interface of the virtual machine;
- **websites**: list of uniform resource locators (URLs) for websites should be visited on each browser

Listing 1 shows an example of an experiment script that visits three pages (called "apache", "nginx-latest" and "nginx-stable") using two distinct browsers (Google Chrome and Mozilla Firefox), repeating each combination five times. Seven experiments were defined for testing purposes, one for each page in the website, including the *gallery-N.php* variations. The *index.php* page contains the text "Hello World" and references an external style sheet resource; the *lorem.php* page contains about eighty thousand characters and does not reference a style sheet. This was done on purpose to confuse the machine learning algorithm when classifying both pages, since the combined length of the *index.php* plus the external style sheet is similar to the length of the single *lorem.php* page, although the number of requested web objects is different. On the other hand,

gallery consists of five pages (*gallery-1.php*, *gallery-2.php*, *gallery-4.php*, *gallery-6.php*, *gallery-8.php*), each showing a different amount of images, which are randomly selected from a collection of 30 images with different dimensions and file sizes.

Web browsers are launched and remotely controlled from Python (or virtually any programming language) using *Selenium* [1], a browser automation framework for web application testing that provides means of automating the most popular web browsers. *Selenium* implements drivers, which are small programs that interact with a specific web browser and perform trivial tasks, such as loading a page given an address, creating and closing tabs or windows, checking if the page has loaded successfully. Additionally, Selenium allows programmers to simulate network conditions (only the Chrome driver, as of the date of writing) and define user preferences for each individual browser, such as disabling *JavaScript* and setting the default download directory. Application testing functionality, such as accessing DOM elements from the page such as verifying the presence of an input field in the page or comparing the style properties of an HTML element is also present, although not relevant for network traffic capture.

The automated capture tool depends on *Tcpdump*, a well-known packet analyzer and capture tool for network traffic that runs under the command line interface. This tool records TCP/IP and other packets being transmitted or received over a network to which the computer is attached. *Tcpdump* works on most Unix-like operating systems, such as Linux or macOS, and relies on the *libpcap* library to capture packets and persist them in binary PCAP files. There is a port of the *Tcpdump* utility for the Windows operating system called *WinDump*, which in turn relies on *WinPcap*, the Windows port of *libpcap*.

The libpcap file format (or PCAP, from "packet capture") format is considered a de-facto standard for network traffic capture among network and security research enthusiasts. Most network analysis tools (such as Wireshark) support PCAP as an input format. This format became the common denominator for storing network capture files in the open-source world. The *libpcap* implementation of the PCAP format has a very simple specification, one of the reasons why this format has gained such a wide usage and recognition. Unfortunately there are quite a few drawbacks and misses some useful features, such as nanosecond time resolution, network interface information and dropped packet statistics.

A graphical user interface is not required to interact with the web browsers. The capture automation tool relies on *PyVirtualDisplay*, a Python wrapper for *Xvfb* that simplifies the task of working with virtual displays within Python applications on Unix-like operating systems. *Xvfb* (X virtual frame buffer) is a display server implementing a virtual *X11* display server protocol that runs applications with graphical user interfaces on virtual memory frame buffers without presenting the actual screen contents or compromising application functionality. Users typically interact with virtual machines using a command line interface (text mode, as opposed to graphical mode). This virtual server does not require the machine to have any kind of graphics adapter, a screen or any input device. Additionally, there is an increasing trend in cloud-based services that provide virtual machines which are meant to accessed remotely and exclusively through SSH.

[1] https://www.seleniumhq.org/

There are some possible improvements for this tool in the future, such as switching to the newer *PCAPNG* file format. *PCAP Next Generation* (PCAPNG) is currently defined as an Internet Engineering Task Force (IETF) draft and aims to bring improvements, extensibility and portability to the industry standard *libpcap* format, such as nanosecond time resolution, per-packet annotations, fields for storing metadata, the ability to capture traffic from multiple network interfaces, and additional dropped packet statistics. *PCAPNG* is currently the default format for *Wireshark* and *TShark*, although the general adoption of this new standard has been slow. The current version of *Tcpdump* does not support writing traffic captures in this format yet.

3.2.2 Analysis tool

Figure 3.2 presents the architecture of the HTTP/2 traffic analysis tool and the interaction between its three subcomponents, as well as its relation with the capture automation tool. The next section discusses the implementation of the parser component, while the visualization and machine learning components will be thoroughly described in the next chapters of this report.

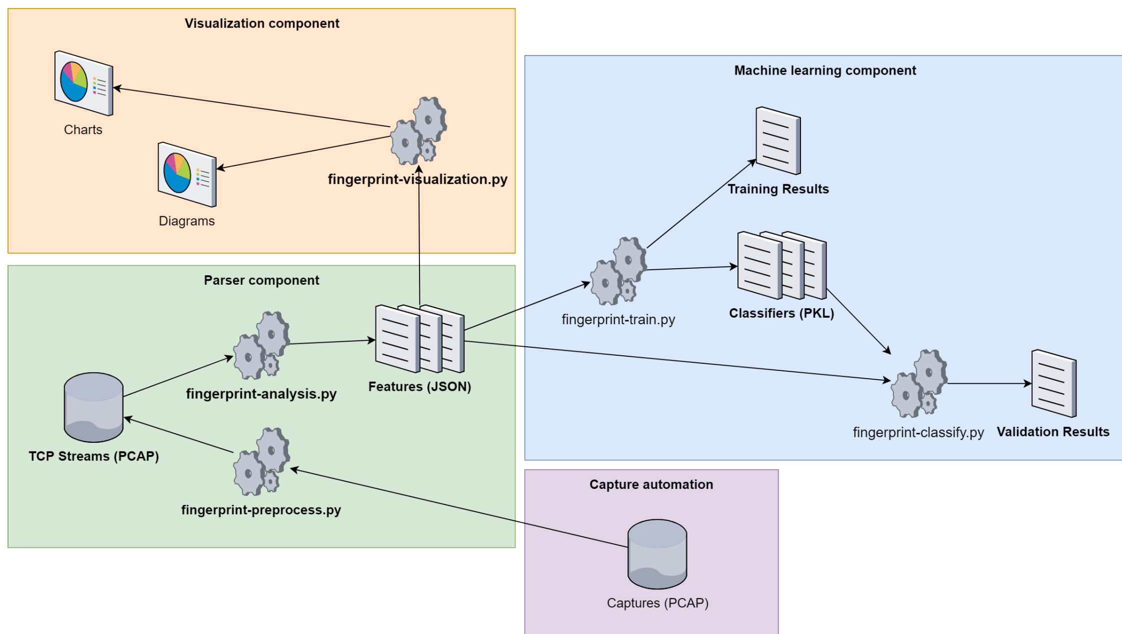

Figure 3.2: Architecture diagram for the HTTP/2 traffic analysis tool

3.3 Parser component

The parser component consists of a script for preprocessing the captures generated by the capture automation tool (*fingerprint-preprocess.py*) and converting them into separate TCP streams, and an application for parsing these files and identifying TLS records, HTTP/2 frames, HTTP/2 streams and web objects (*fingerprint-analysis.py*). The preprocessor iterates through the PCAP files, enumerating the TCP streams from each capture. Since the packet dissector from *TShark*

and *Wireshark* could not read the entire capture as a whole and properly decrypt the TLS records due to implementation bugs, the individual TCP streams are extracted to separate files. The solution was to generate a byte dump for each TCP stream in text format with the contents of every packet sent or received in that stream, and then reassemble the packets by using *txt2pcap* to convert the byte dump into a PCAP file. After it has finished processing all TCP streams from the original PCAP, the preprocessor script also writes a YAML file with information regarding the TCP streams that were identified, such as the client and server addresses and the stream identifier.

The parser component reads the PCAP files generated by the preprocessor and calls *Tshark* to export the packets to *PDML* (Packet Description Markup Language) before being fed into the parser. The PDML format has a syntax similar to XML and describes the TCP/IP layer, TCP segments and TLS records for each packet. If that particular capture has the pre-master secrets file supplied, *TShark* also attempts to decrypt the TLS records and exports information regarding the HTTP/2 frames obfuscated on each record. There is an advantage of using this format over parsing binary PCAP files directly since *TShark* already performs partial analysis on the data, such as identifying TLS segments and unpacking HTTP/2 headers, which saved a great amount of effort during the implementation of the parser component.

The PDML file is then parsed using the *ElementTree* library that reads the XML tree, supported by custom parsing functions for each data type (e.g. integers, strings, booleans) that validate and convert the raw XML data into Python objects. The parser component outputs several JSON files with capture statistics, packet information, TLS record information, and HTTP/2.0 frame and stream information. Different Python classes define the structure of each type of object that the parser can read from the PDML file, and how to serialize that data when exporting to JSON, and the formatting for each field. For instance, enumeration types are converted to their respective string representation instead of the numeric value stored in memory.

The parser component also simulates the HTTP/2 connection state machine and attempts to perform TLS record to HTTP/2 frame mapping. Since HTTP/2 frames may appear segmented over multiple records, the parser component identifies these records and classifies how each frame appears inside a record. A frame that is "unique" for a given record has no segments in other records, a "reassembled frame" has their last segment obfuscated inside that record, and a "partial frame" means that the record contains the first or a middle segment of that frame.

Figure 3.3 shows the structure of the capture directory after running the *fingerprint-preprocess.py* script, followed by the *fingerprint-analysis.py* to parse information from TCP streams. Experiments are placed at the top level of the capture directory, in subdirectories that correspond to the "id" field of the experiment script. Inside each experiment directory there are three subdirectories ("apache", "nginx-latest" and "nginx-stable"), one for each web engine being tested. Inside the engine directory are subdirectories for every web browser, iteration and TCP stream combination. For instance, "firefox-R2-S3" means that this directory contains traffic features for TCP stream number 3 from the second capture performed on Mozilla Firefox for that particular web engine. Every feature directory contains six JSON files: overall statistics (*features.json*), information regarding HTTP/2 frames (*frames.json*), HTTP/2 web objects (*objects.json*), packets (*packets.json*),

TLS records (*records.json* and HTTP/2 streams (*streams.json*).

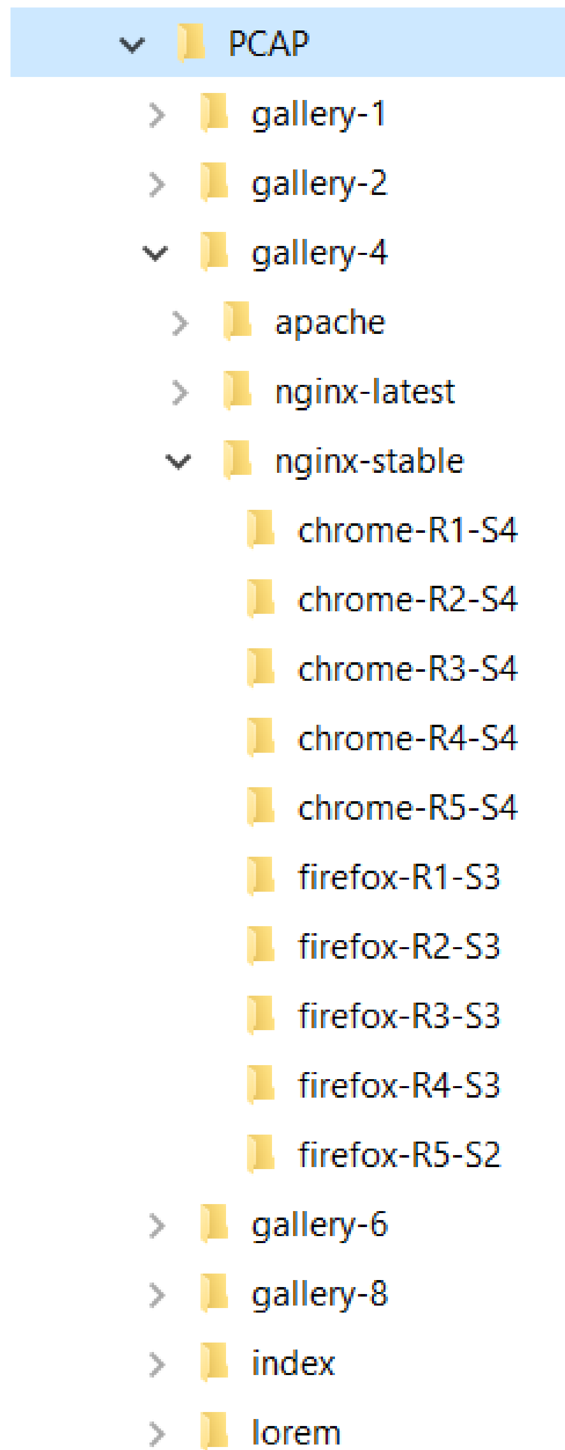

Figure 3.3: Example tree of the capture directory after running both *fingerprint-preprocess.py* and *fingerprint-analysis.py* scripts.

Chapter 4

Visualization component

The visualization component was designed as a proof of concept application to demonstrate the potential of the comprehensive output generated by the parser component. It employs several Python plotting and visualization libraries to render charts and export them to image and vector graphics formats. There are three main kinds of visualizations:

- **sequence**: horizontal bar charts representing sequences of parallel events, similar to *Gantt* charts used for project management and task schedule; the tool generates two distinct visualizations for this type: "*streams*", which displays the sequence of frames (and their type) for each HTTP/2 stream; "*objects*", which shows the succession of "request headers", "response headers", "payload begin" and "payload end" events for every web object requested by the web browser, including the HTML itself.
- **charts**: bar charts, stacked charts, pie charts, X/Y scatter and box-and-whisker plots for different features of HTTP/2 frames, such as frame type count, frame length distribution and frame length to record length mapping;
- **graph**: a stream dependency graph, where HTTP/2 streams are represented as graph nodes and dependencies between two streams are represented as directed edges.

4.1 Implementation

The sequence charts depend on custom code written for *Matplotlib* [kn:18a], more specifically the *pyplot* module provided by the same package. *Matplotlib* is a plotting library for the Python programming language. It provides an object-oriented application programming interface for embedding plots into applications, while *Pyplot* is a collection of command style functions that make *Matplotlib* work similarly to *MATLAB*, the industry standard development environment for numerical computing.

```
def plot_gantt(items, colors):

  labels = []
  _, axis = pyplot.subplots(figsize=(6, 3))
  df = pandas.DataFrame(list(items), columns=GANTT_COLUMNS)
  df["Diff"] = df.Finish - df.Start

  for index, task in enumerate(df.groupby("Task")):

    labels.append(task[0])

    for resource in task[1].groupby("Resource"):

      axis.broken_barh(
        resource[1][["Start", "Diff"]].values,
        (index - 0.4, 0.8),
        color=colors[resource[0]]
      )

  axis.set_yticks(range(len(labels)))
  axis.set_yticklabels(labels)
  axis.set_xlabel("time [ms]")
  pyplot.tight_layout()
  pyplot.show()
```

Listing 2: Function for plotting a Gantt-like chart given a set of tasks and their respective durations.

The stream dependency graphs are generated using *NetworkX* [kn:18b], a Python library for creating, manipulating and studying the structure, dynamics, and functions of complex networks. *NetworkX* can create and display both undirected graphs and directed graphs, convert between several graph formats, analyze network structure, and perform several operations on graphs such as finding subgraphs, cliques, and exploring adjacency, in-degree and out-degree.

```
def insert_dependency(self, stream, parent):
  self.graph.add_edge(f"S{stream}", f"S{parent}")
```

Listing 3: Function that inserts a dependency (connects a directed edge) between a given stream and its parent stream nodes.

```
def insert_server_stream(self, stream):
  stream_id = f"S{stream}"
  self.labels[stream_id] = stream_id
  self.graph.add_node(stream_id)
```

Listing 4: Function that inserts a server-initiated stream as a node graph.

```
def plot_dependency_graph(self, instance):

  for stream_id, stream in instance["streams"].items():

    if stream["direction"] == "S2C":
      self.insert_server_stream(stream_id)
    elif stream["direction"] == "C2S":
      self.insert_client_stream(stream_id)

    if stream["parent"]:
      self.insert_dependency(stream_id, stream["parent"])

  layout = nx.nx_pydot.pydot_layout(self.graph, prog="dot")
  nx.draw_networkx_nodes(self.graph, layout)
  nx.draw_networkx_edges(self.graph, layout)
  nx.draw_networkx_labels(self.graph, layout, self.labels)
  plt.show()
```

Listing 5: Function that generates a stream dependency graph from the *streams.json* file generated by the parser component.

4.2 Results

This section presents some charts generated using the visualization component and attempts to identify HTTP/2 protocol and network packet features that could help machine learning algorithms infer about the contents of encrypted SSL records such as the count, type or even the length of individual HTTP/2 frames obfuscated inside these records. The charts were generated using the same dataset, which consisted of all captures obtained from each of the three experiments (“gallery”, “index” and “lorem”).

4.2.1 Web objects sequence

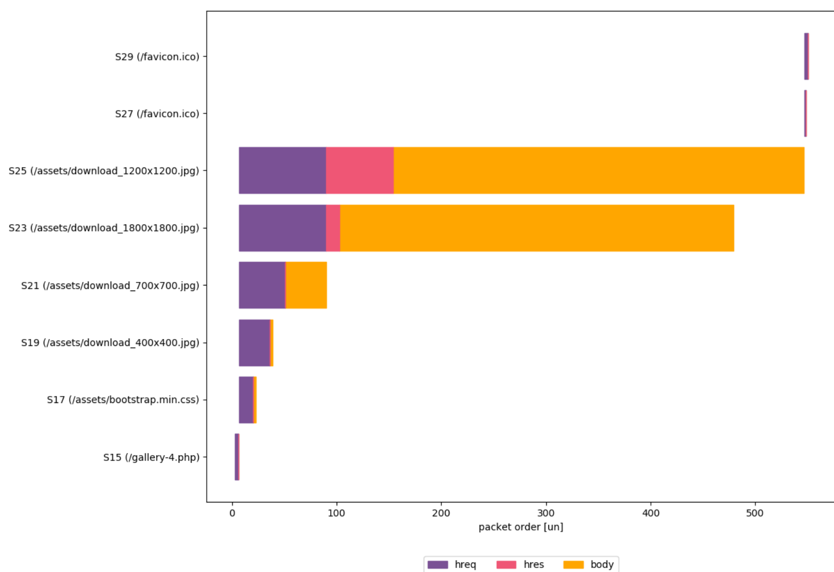

Figure 4.1: Sequence of events for web objects requested by the browser

The web objects sequence chart (Figure 4.1) shows the interval (number of packets) between the browser sending the request for a particular object and the server sending the matching response *HEADERS* (represented by the purple bars); pink bars represent the delta between the client receiving the response *HEADERS* and the first *DATA* frame containing the binary payload for the object sent by the server, and lastly, the interval between the first and last *DATA* frames of the binary payload is represented by the orange bars. The x-axis represents the packet sequence number, allowing to reconstruct the succession of packets and records without a time scale. *gallery-4.php* was the first object requested by the browser, and both *favicon.ico* objects were the last.

4.2.2 Stream dependency graph

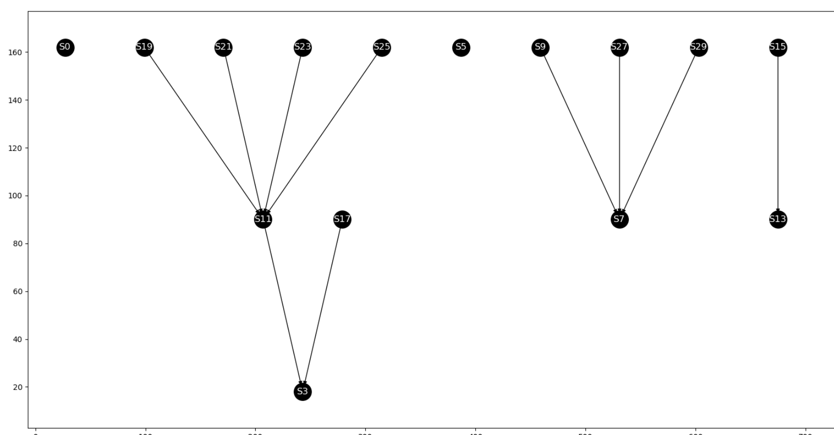

Figure 4.2: Stream dependency graph

The stream dependency graph shows dependencies between HTTP/2 streams (graph nodes). In this particular example (Figure 4.2), the node representing stream 0 (S0) has neither incoming nor outgoing edges. This behavior is expected, as S0 is always reserved for connection control messages and cannot be used to create other streams. On the other hand, streams S19, S21, S23 and S25 have outgoing edges to S11, which in turn points to S3. This means that the first set of streams have S11 as a direct dependency, and S11 depends on S3. The stream upon which a stream is dependent is called a **parent stream**. For instance, S3 is the parent of both S11 and S17. Dependent streams should only allocate resources if either all of the streams that it depends on (the chain of parent streams) are closed or it is not possible to make further progress on them.

4.2.3 Length distribution

Since website payload data has to be segmented and transmitted inside smaller network packets, *DATA* frames were the most common type (Figure 4.3), followed by *HEADERS* and *WINDOW_UPDATE*, while *GO_AWAY* and *RST_STREAM* were the less common types. There is a clear distinction between *DATA* frames and the remaining types for lengths greater than 200 bytes. The remaining types have lengths ranging from 0 bytes to 20 bytes, and *MAGIC* frames were the only type that could be identified unequivocally for this experiment due to their fixed length.

The most significant difference in the client frames distribution (Figure 4.4) relatively to the overall frame distribution chart (Figure 4.3) is that the client does not send *DATA* frames. Moreover, lengths greater than 200 bytes were not observed, and all the frames larger than 20 bytes were of the *HEADERS* type. *GO_AWAY*, *WINDOW_UPDATE*, *PRIORITY* and *RST_STREAM* frames never exceeded 10 bytes. Only *MAGIC* and *SETTINGS* frames could be unequivocally identified since they are characterized by lengths between 10 and 20 bytes.

Looking at the frames sent by the server (Figure 4.5), the majority was of the *DATA* type and had lengths in the range of 800-1024 bytes or between 6144-8192 bytes. Further observations showed that the first peak of *DATA* frames were primarily sent by *Apache* servers, while the latter peak corresponds to frames sent by the *nginx* servers. *DATA* frames were the less trivial type to identify, although there was a clear separation between them and the remaining types for frame lengths greater than 100 bytes. For similar reasons, *HEADERS* were the second less trivial frame type, since their length can vary from 10 to 100 bytes. On the other hand, *GO_AWAY* frames had lengths comprised between 10 and 20 bytes and could be unequivocally identified, as well as WINDOW_UPDATE frames, which had lengths up to 10 bytes.

Regarding the distribution of the first frames in each stream (Figure 4.6), the number of types is narrowed down to three: *HEADERS*, *PRIORITY* and *SETTINGS*, and each type falls inside its own length bucket. *PRIORITY* frames are under 10 bytes and have fixed length (zero variance), *SETTINGS* frames are also of fixed length and fall in the 10-20 bytes category, while the length of *HEADERS* frames can vary between 20 and 200 bytes. Thus, the mapping between frame lengths and types for the first frame of each stream seems trivial.

Lastly, the distribution of the last frames in each stream (Figure 4.7) shows a different scenario: five frame types, three of them having fixed length and falling in the first histogram class, *GO_AWAY* frames having lengths up to 20 bytes and falling in the first two bins, while the rest of the histogram is populated with *DATA* frames ranging from 40 to 8192 bytes. The box plot does not show the length distribution of *DATA* frames, since that would make analysis difficult for the remaining types (there is a huge length gap between *DATA* and the other frame types).

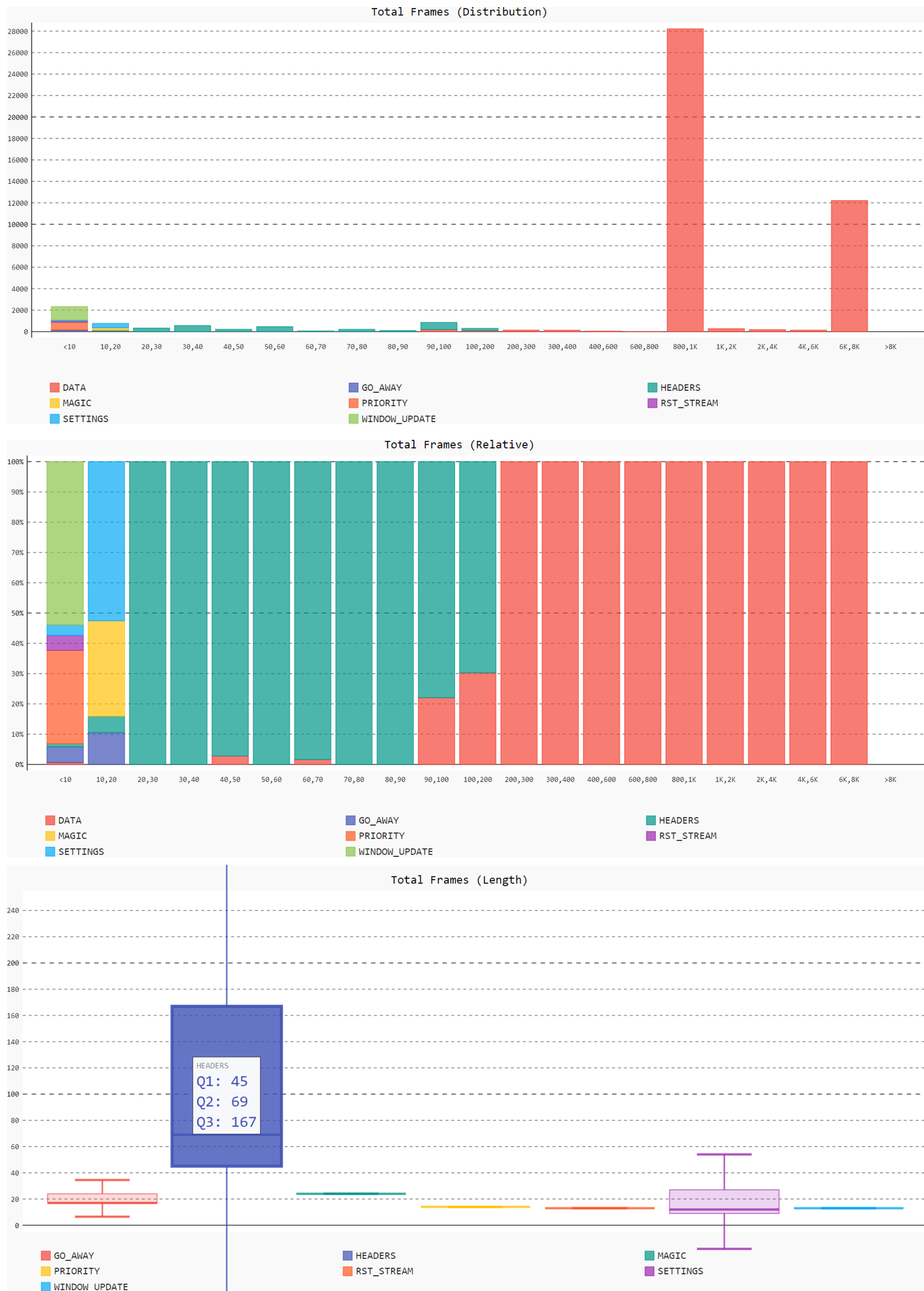

Figure 4.3: Length distribution of frames (absolute and relative units, box-and-whisker plot)

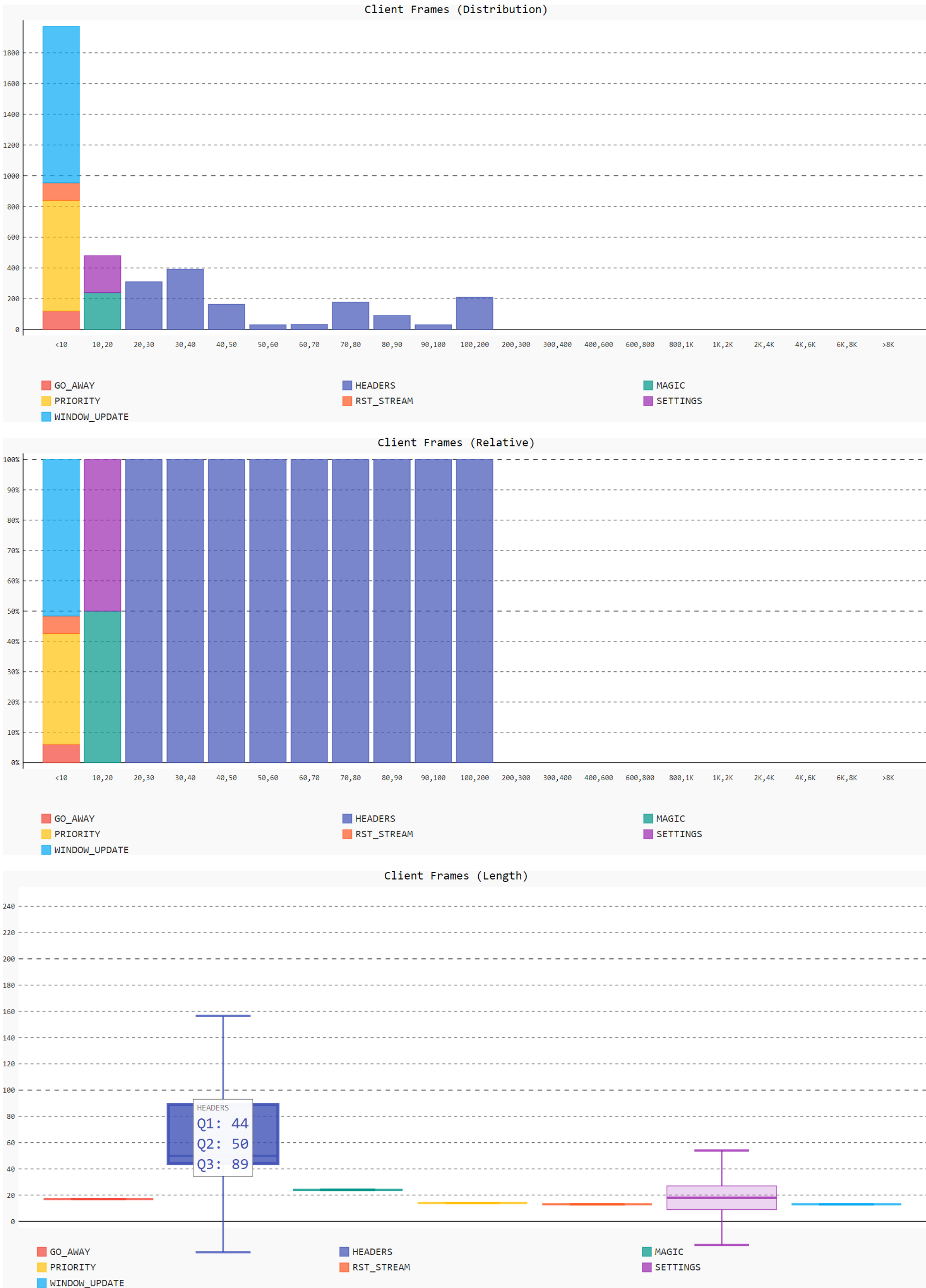

Figure 4.4: Length distribution of frames sent by the client (absolute and relative units, box-and-whisker plot)

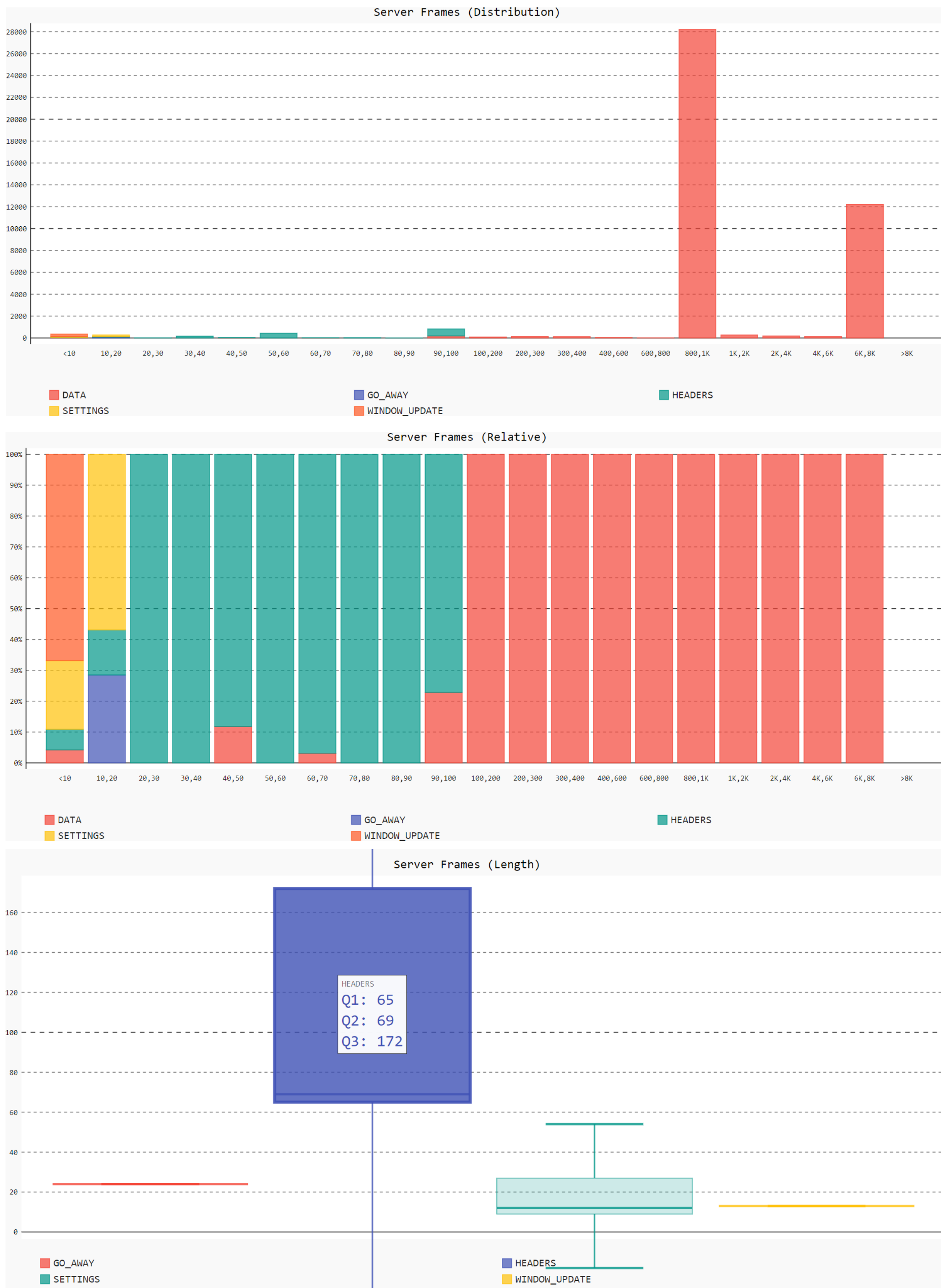

Figure 4.5: Length distribution of frames sent by the server (absolute and relative units, box-and-whisker plot)

Figure 4.6: Length distribution of the first frames from each stream (absolute and relative units, box-and-whisker plot)

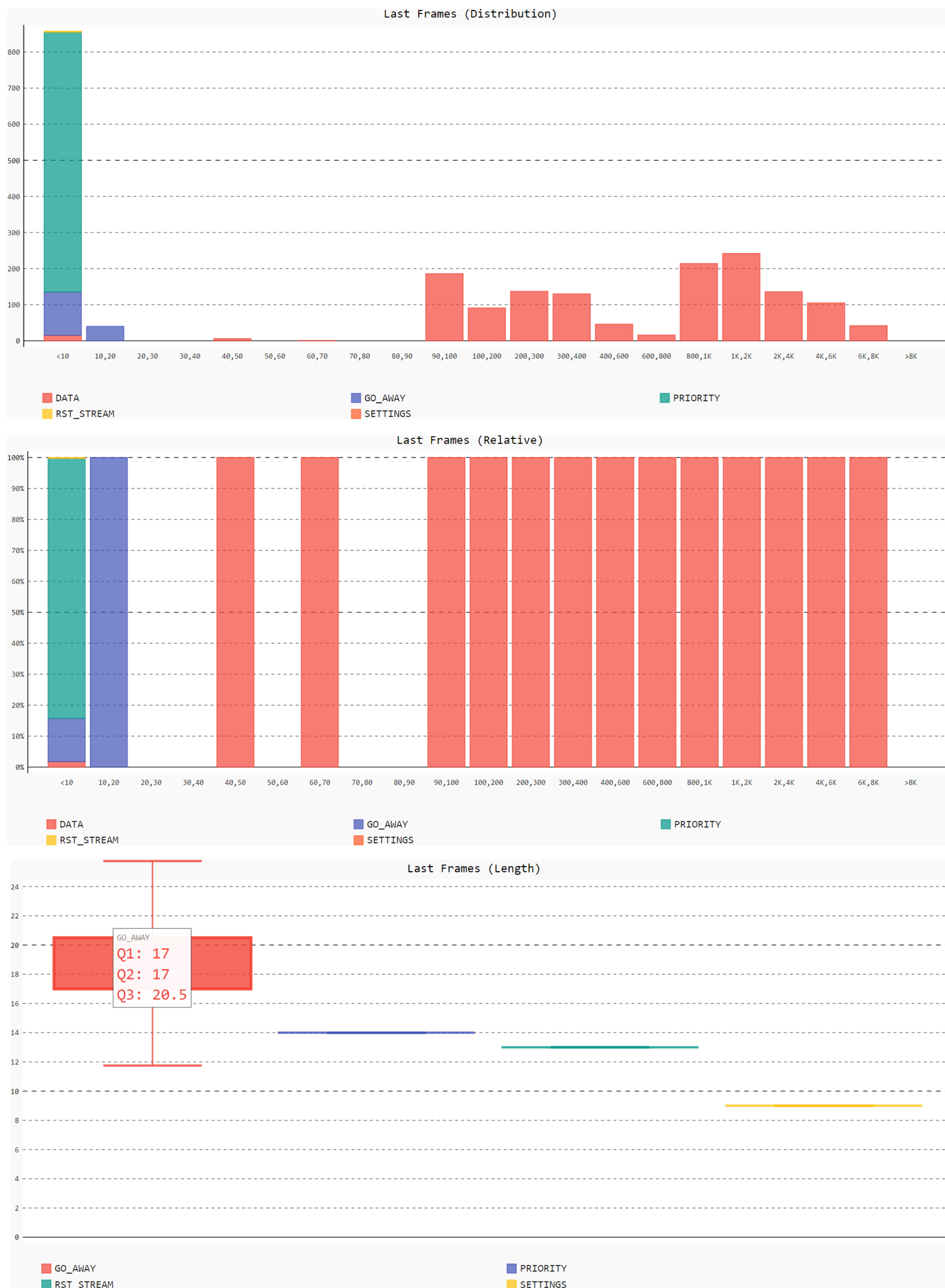

Figure 4.7: Length distribution of the last frames from each stream (absolute and relative units, box-and-whisker plot)

4.3 Discussion

The charts provided by the visualization component helped identify unique features of HTTP/2 client-server communication and HTTP/2 frame characteristics, such as their length distribution, average count by type and average length by type. The current implementation generates the same length distribution visualizations for the entire set of frames being analyzed, frames sent by the client, frames sent by the user, the first frame within each stream and the last frame within each stream. Due to the diversity of charts already implemented by this component, extending its functionality in order to visualize new traffic features or study the correlation between two features is trivial.

The dependency graph and sequence charts are only suitable for small captures, up to 20 streams or web objects. More complex captures may generate denser graphs that are hard to analyze manually due to visual pollution. This could be overcome by adopting a more minimalistic, less “polluted” layout or by filtering out streams that do not have web objects associated and streams containing objects that do not meet a given minimum length or content type, thus reducing the number of streams being visualized for analysis.

Chapter 5

Machine learning component

The machine learning component consists of a training and a classification tool. The training tool takes website captures preprocessed by the parser component, trains classifiers and persists the trained model for each classifier in the local file system for future fitting and classification. Training statistics for each classifier are computed and written to YAML files. The classification tool loads the trained models generated by the training tool and executes predictions on previously unseen samples, checks these predictions against their true values, generates the confusion matrix for each classifier and writes results to YAML files for further analysis and comparison. Features are extracted from metadata at both packet level (direction, length, number of records, order) and TLS record level (length and position/order inside the packet), on encrypted application data records. Three classifiers were implemented and are the following:

- **ehes**: indicates if the record contains HTTP/2 frames with the *END_HEADERS* flag set, but not *END_STREAM* ("eh"), just the *END_STREAM* flag, but not *END_HEADERS* ("es"), both flags ("ehes") or none of the flags ("none"); however, there is no way to tell apart a record containing a single HTTP/2 frame that sets both *END_HEADERS* and *END_STREAM* flags from a record containing multiple frames, in which one sets the *END_HEADERS* flag and another sets the *END_STREAM* flag exclusively.
- **frame_count**: indicates the number of HTTP/2 frames obfuscated inside a given record; to avoid class explosion and imbalance, records containing four or more frames fall into the same category, here labeled as "4+"; other possible classes include "0", "1", "2" and "3".
- **frame_type**: indicates the type of frames obfuscated inside a given record; as a limitation, and to avoid class imbalance, this classifier only infers about the presence or absence of a given type; classes are represented by the respective frame types, separated by commas, alphabetically ordered; for instance, "PRIORITY,SETTINGS,WINDOW_UPDATE" indicates the presence of frames from the three types inside the record, not the actual count of each type.

5.1 Implementation

The training tool allows to choose an experiment and an input directory containing website captures, then extracts the TLS records found on each capture into a single dataset and feeds them to five machine learning algorithms (decision trees, gradient boosting, K-nearest neighbors, random forest and support vector machines). This dataset is then randomly split into training and test sets and cross-validation is performed in order to select the machine learning algorithm providing the best overall accuracy for each classifier.

The training tool also allows generating datasets from a subset of captures. The available filters include "sample", which randomly selects N captures from the entire set of experiments generated by the capture automation tool; "browser", that only selects captures generated by a given web browser, "engine", which only selects captures generated from the website running on a given web engine. The results in the next section make use of nine subsets: "browser-chrome" and "browser-firefox", that only contain captures generated by Google Chrome and Mozilla Firefox browsers respectively, "engine-apache", "engine-nginx-latest" and "engine-nginx-stable", which contain captures generated from the website running on Apache, NGINX 1.14.0 and NGINX 1.10.3 web engines respectively, and lastly "sample-0", "sample-25", "sample-50" and "sample-100", that contain the entire set of captures, a randomly selected subset of 25 (about 12.5% of the total number of captures), 50 (25%) and 100 (50%) captures respectively.

The classification tool recursively walks through a specific directory and attempts to find valid captures. To avoid duplicates, every capture has a unique hash and the classification tool checks it against the captures that were previously selected as the training dataset. If the output files generated by the parser component for a given capture are valid, it then proceeds to execute each three of the implemented classifiers for every TLS record found in that capture and writes the validation results to a YAML file, one per each capture, containing the following parameters for each classifier: accuracy, confusion matrix, F-score, average F-score, number of correctly classified records, number of incorrectly classified records, precision, average precision, recall, average recall, support and total number of records found. The results file also contains a list of incorrectly classified samples, comparing the predicted values with the true class values, along with the values of the features for that sample as reference for manual analysis.

After classifying individual captures, the tool concatenates the samples from each capture and executes the classifier against the newly generated dataset. This will give an overview of the classification and validation results for the whole set of captures inside the test directory. The global classifier then generates a *validation.yml* file containing similar statistics as the individual results file, plus a breakdown of correctly and incorrectly classified records for each capture. A record is considered to be correct if all three classifiers predicted correctly, and incorrect if at least one of the classifiers had prediction errors.

All the specific algorithms, statistics, metrics and auxiliary functions used in the machine learning component are found in Scikit-learn [kn:18c], an open-source machine learning library for the Python programming language that provides efficient implementations of multiple classifi-

cation, regression (supervised learning) and clustering algorithms (unsupervised learning) including support vector machines, random forests, decision trees and *k-means*. Scikit-learn has a clean, uniform and streamlined programming interface that makes switching to a new model or algorithm a straightforward procedure, and provides comprehensive online documentation and tutorials for learning about machine learning algorithms, as well as data visualization.

5.2 Results

	CART	**Gradient**	**K-Nearest**	**Random Forest**	**SVM**
ehes	98.4% ± 0.22%	**98.6% ± 0.17%**	98.4% ± 0.21%	98.5% ± 0.24%	97.9% ± 0.26%
frame_count	96.9% ± 0.26%	**97.5% ± 0.23%**	97.2% ± 0.23%	97.1% ± 0.18%	97.2% ± 0.24%
frame_types	99.3% ± 0.09%	99.1% ± 0.03%	99.0% ± 0.10%	**99.3% ± 0.09%**	98.8% ± 0.15%
Average	98.2%	98.4%	98.2%	98.3%	98.0%

Table 5.1: Cross-validation accuracy metrics for five distinct machine learning algorithms (best accuracy values for each classifier in bold) using samples from “sample-0”.

The purpose of the first experiment was to survey multiple machine learning algorithms provided by *Scikit* and evaluate their performance for the entire set of captures. The accuracy values for cross-validation were very similar among the five selected algorithms (Table 5.1), and none particularly stood out in this comparison. Support vector machines (SVM) had the worst accuracy of all four by a difference of 0.4% to the best performing algorithm, and were also the second slowest next algorithm to gradient boosting, which had the best accuracy overall. Decision trees and random forest (ensemble-based learning) provided the best balance between accuracy and execution time.

	sample-25			**sample-50**		
	ehes	frame_count	frame_types	ehes	frame_count	frame_types
Correct	687	664	695	2225	2206	2247
Incorrect	19	42	11	35	54	13
Accuracy	97.3%	94.1%	98.4%	98.5%	97.6%	99.4%

Table 5.2: Number of correctly and incorrectly classified samples and classification accuracy for the cross validation results using samples from “sample-25” and “sample-50”.

	sample-100			**sample-0**		
	ehes	frame_count	frame_types	ehes	frame_count	frame_types
Correct	3210	3195	3253	7843	7737	7891
Incorrect	69	84	26	115	221	67
Accuracy	97.9%	97.4%	99.2%	98.6%	97.2%	99.2%

Table 5.3: Number of correctly and incorrectly classified samples and classification accuracy for the cross validation results using samples from “sample-100” and “sample-0”.

A second experiment was performed in order to evaluate the impact of the number of samples used for training in the cross-validation results (Table 5.3). Four datasets were generated, with samples from 25 randomly selected captures ("sample-25"), 50 captures ("sample-50") and 100 captures ("sample-100"), as well as every capture in the dataset "sample-0"), respectively. The results were not very conclusive, as one would expect the classifier accuracy to improve as the number of training samples increases, but since the website captures were not diversified, inserting more samples from similar captures only changes the accuracy results by a few percentage points.

5.2.1 Web Engine - Apache

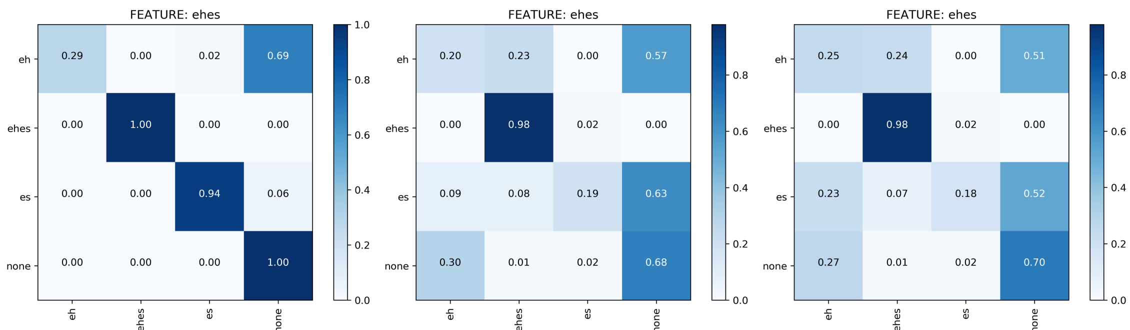

Figure 5.1: Confusion matrices for the "engine-apache" experiment, "ehes" classifier: cross-validation results (Apache *versus* Apache), prediction results using samples from "engine-nginx-latest" (Apache *versus* NGINX 1.14.0), prediction results using samples from "engine-nginx-stable" (Apache *versus* NGINX 1.10.3).

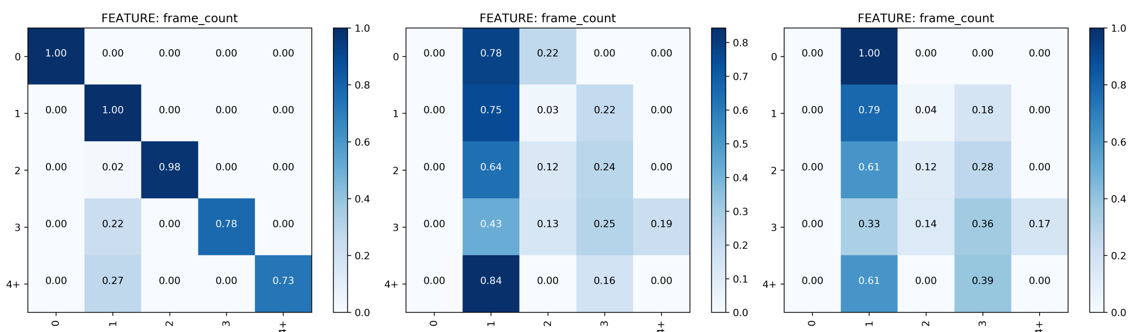

Figure 5.2: Confusion matrices for the "engine-apache" experiment, "frame_count" classifier: cross-validation results (Apache *versus* Apache), prediction results using samples from "engine-nginx-latest" (Apache *versus* NGINX 1.14.0), prediction results using samples from "engine-nginx-stable" (Apache *versus* NGINX 1.10.3).

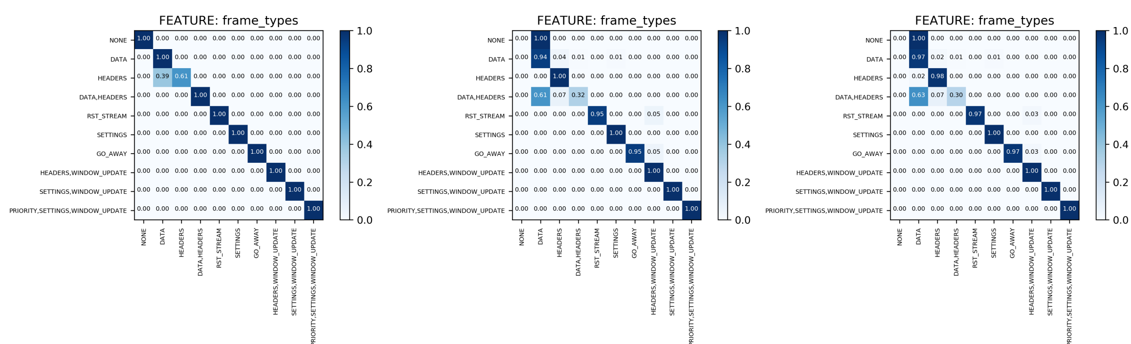

Figure 5.3: Confusion matrices for the “engine-apache” experiment, “frame_types” classifier: cross-validation results (Apache *versus* Apache), prediction results using samples from “engine-nginx-latest” (Apache *versus* NGINX 1.14.0), prediction results using samples from “engine-nginx-stable” (Apache *versus* NGINX 1.10.3).

The classifiers trained for the “engine-apache” experiment could not accurately classify the frame count for the capture samples using other web engines. This means that there are noticeable differences between *Apache* and *NGINX* when it comes to the distribution of HTTP/2 frames among TLS records, primarily frames of smaller lengths such as the *PRIORITY*, *HEADERS* and *WINDOW_UPDATE* types.

	Apache *versus* Apache		
	ehes	frame_count	frame_types
Correct	6142	6173	6147
Incorrect	42	11	37
Accuracy	99.3%	99.8%	99.3%

Table 5.4: Number of correctly and incorrectly classified samples and classification accuracy for cross-validation using samples from “engine-apache”.

	Apache *versus* NGINX 1.14.0			**Apache *versus* NGINX 1.10.3**		
	ehes	frame_count	frame_types	ehes	frame_count	frame_types
Correct	3090	1432	4127	2938	1284	3898
Incorrect	1503	3161	466	1340	2994	380
Accuracy	67.3%	31.1%	89.9%	68.7%	30.0%	91.1%

Table 5.5: Number of correctly and incorrectly classified samples and prediction accuracy using samples from “engine-nginx-latest” and “engine-nginx-stable”; the three classifiers were trained using samples from *engine-apache*.

Regarding *END_HEADERS* and *END_STREAM* presence (Figure 5.1), the predictions made on the NGINX 1.10.3 (*nginx-stable*) experiments provided better results than the NGINX 1.14.0 (*nginx-latest*) samples accuracy wise. Looking at the confusion matrix generated for the “ehes”

classifier, only 29% of the frames having the “END_HEADERS” flag were classified correctly as “eh”, while the majority of them (around sixty-nine percent), were classified as “none” and the remaining as “es”. Regarding frame types (Figure 5.3), most samples belonging to the “NONE” and “DATA,HEADERS” classes were incorrectly classified as “DATA” for both “nginx-stable” and “nginx-latest” experiments, but overall the classification accuracy for the other types ranged from 94% to 100%.

5.2.2 Web Engine - NGINX 1.14.0

	NGINX 1.14.0 *versus* NGINX 1.14.0		
	ehes	frame_count	frame_types
Correct	887	813	905
Incorrect	32	106	14
Accuracy	96.5%	88.4%	98.5%

Table 5.6: Number of correctly and incorrectly classified samples and classification accuracy for cross-validation using samples from the “engine-nginx-stable” experiment.

	NGINX 1.14.0 *versus* Apache			**NGINX 1.14.0 *versus* NGINX 1.10.3**		
	ehes	frame_count	frame_types	ehes	frame_count	frame_types
Correct	29720	11086	28240	4079	3854	4184
Incorrect	1198	18732	2678	199	424	94
Accuracy	96.1%	35.9%	91.3%	95.3%	89.6%	97.8%

Table 5.7: Number of correctly and incorrectly classified samples and prediction accuracy using samples from “engine-apache” and “engine-nginx-stable”; the three classifiers were trained using samples from *engine-nginx-latest*.

The “frame_count” classifier trained for the “engine-nginx-latest” experiment could not predict the majority of samples from the “engine-apache” experiment (Table 5.7). 11086 records were classified correctly and 18732 were misclassified out of 29818 records, resulting in an accuracy of just 35.9%. This shows the contrast between Apache and NGINX web engines. On the other hand, the same classifier trained with samples from the “engine-apache” experiment showed an accuracy of 31.1% when attempting to classify the samples from the “engine-nginx-latest” experiment.

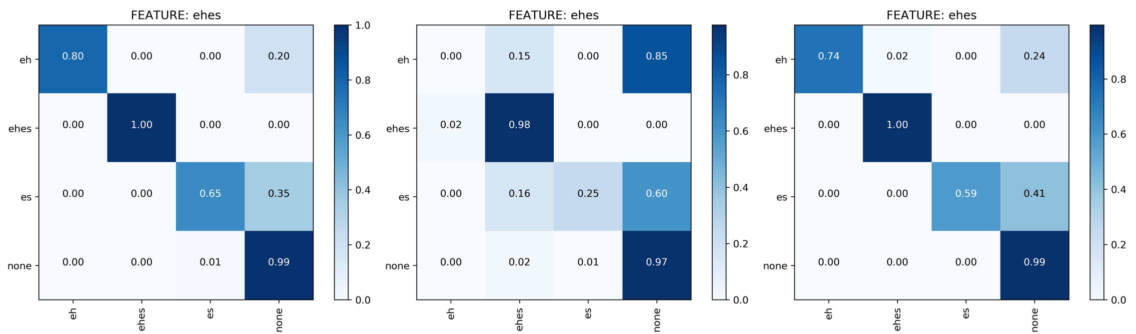

Figure 5.4: Confusion matrices for the “engine-nginx-latest” experiment, “ehes” classifier: cross-validation results (NGINX 1.14.0 *versus* NGINX 1.14.0), prediction results using samples from “engine-apache” (NGINX 1.14.0 *versus* Apache), prediction results using samples from “engine-nginx-stable” (NGINX 1.14.0 *versus* NGINX 1.10.3).

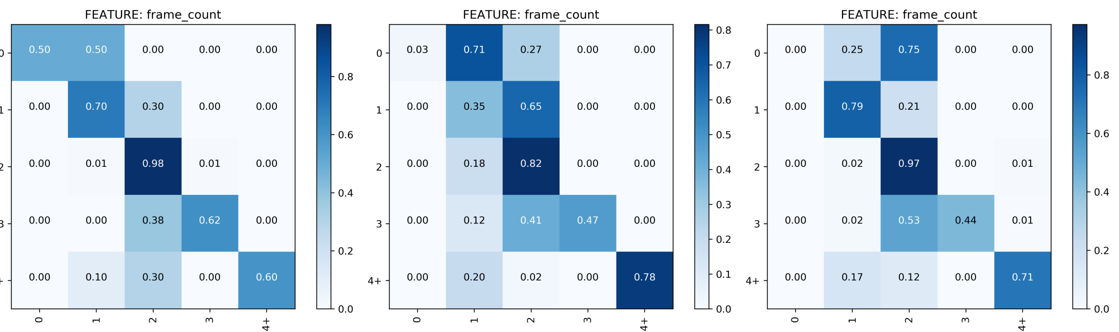

Figure 5.5: Confusion matrices for the “engine-nginx-latest” experiment, “frame_count” classifier: cross-validation results (NGINX 1.14.0 *versus* NGINX 1.14.0), prediction results using samples from “engine-apache” (NGINX 1.14.0 *versus* Apache), prediction results using samples from “engine-nginx-stable” (NGINX 1.14.0 *versus* NGINX 1.10.3).

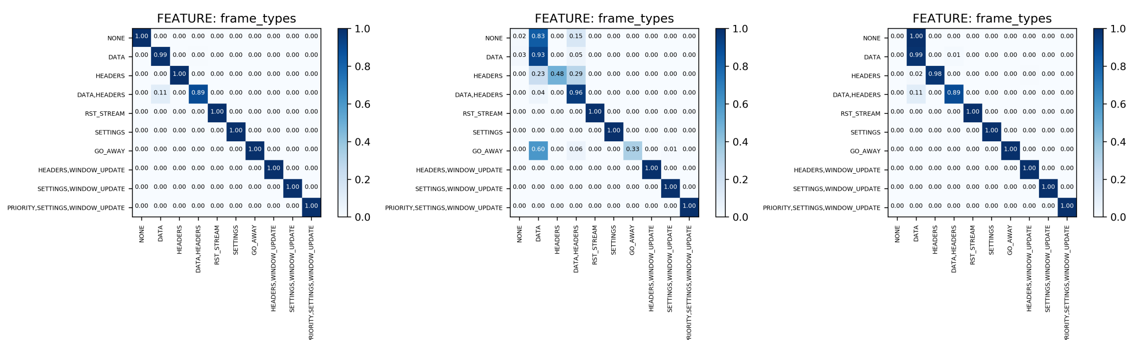

Figure 5.6: Confusion matrices for the “engine-nginx-latest” experiment, “frame_types” classifier: cross-validation results (NGINX 1.14.0 *versus* NGINX 1.14.0), prediction results using samples from “engine-apache” (NGINX 1.14.0 *versus* Apache), prediction results using samples from “engine-nginx-stable” (NGINX 1.14.0 *versus* NGINX 1.10.3).

The predictions made on website captures from the “engine-nginx-stable” tell a different story. There are negligible differences between both NGINX versions, as shown by the confusion matrices (Figures 5.4, 5.5 and 5.6) and cross-validation results (Table 5.6). The majority of frames with *END_HEADERS* or *END_STREAMS* was correctly identified. Only the *NONE* class in the “frame_types” classifier showed a deviation from the cross-validation results. The samples from the “engine-apache” experiment had lower accuracy but both classifiers were still able to correctly predict a great amount of samples.

5.2.3 Web Engine - NGINX 1.10.3

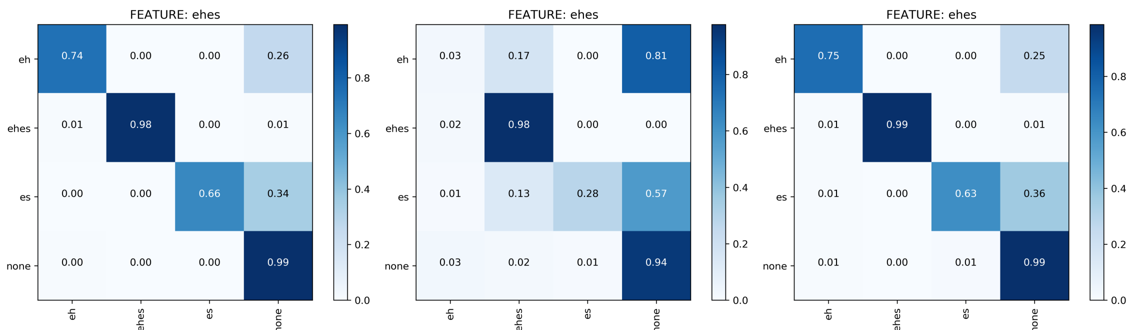

Figure 5.7: Confusion matrices for the “engine-nginx-stable” experiment, “ehes” classifier: cross-validation results (NGINX 1.10.3 *versus* NGINX 1.10.3), prediction results using samples from “engine-apache” (NGINX 1.10.3 *versus* Apache), prediction results using samples from “engine-nginx-latest” (NGINX 1.10.3 *versus* NGINX 1.14.0).

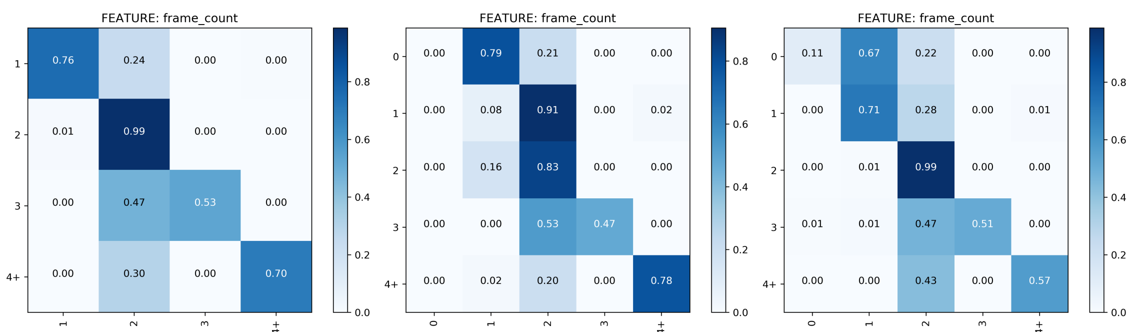

Figure 5.8: Confusion matrices for the “engine-nginx-stable” experiment, “frame_count” classifier: cross-validation results (NGINX 1.10.3 *versus* NGINX 1.10.3), prediction results using samples from “engine-apache” (NGINX 1.10.3 *versus* Apache), prediction results using samples from “engine-nginx-latest” (NGINX 1.10.3 *versus* NGINX 1.14.0).

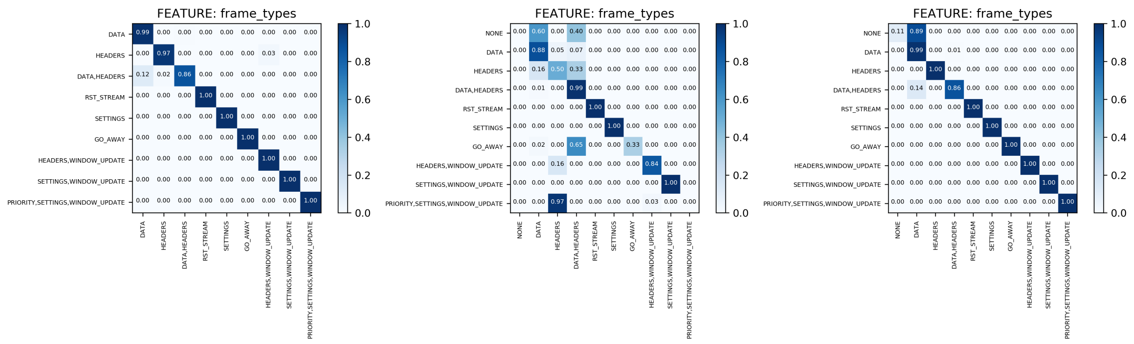

Figure 5.9: Confusion matrices for the "engine-nginx-stable" experiment, "frame_types" classifier: cross-validation results (NGINX 1.10.3 *versus* NGINX 1.10.3), prediction results using samples from "engine-apache" (NGINX 1.10.3 *versus* Apache), prediction results using samples from "engine-nginx-latest" (NGINX 1.10.3 *versus* NGINX 1.14.0).

	NGINX 1.10.3 *versus* NGINX 1.10.3		
	ehes	frame_count	frame_types
Correct	819	774	841
Incorrect	37	82	15
Accuracy	95.7%	90.4%	98.2%

Table 5.8: Number of correctly and incorrectly classified samples and classification accuracy for cross-validation using samples from "engine-nginx-stable".

	NGINX 1.10.3 *versus* Apache			**NGINX 1.10.3 *versus* NGINX 1.14.0**		
	ehes	frame_count	frame_types	ehes	frame_count	frame_types
Correct	28736	2824	26799	4384	4057	4500
Incorrect	2182	28094	4119	209	536	93
Accuracy	92.9%	9.1%	86.7%	95.4%	88.3%	98.0%

Table 5.9: Number of correctly and incorrectly classified samples and prediction accuracy using samples from "engine-apache" and "engine-nginx-latest"; the classifiers were trained using samples from *engine-nginx-stable*.

The "engine-nginx-stable" experiment shows similar results as "engine-nginx-latest". The "frame_count" classifier could not predict the majority of samples from the "engine-apache" experiment (Table 5.9). 2824 records were classified correctly and 28094 were misclassified out of 30918 records, resulting in an accuracy of just 9.1%. On the other hand, the same classifier trained with samples from the "engine-apache" experiment showed an accuracy of 33.0% when attempting to classify against samples from the quotesengine-nginx-latest experiment. There are negligible differences between both NGINX versions, as shown by the confusion matrices for each classifier (Figures 5.7, 5.8 and 5.9) and cross-validation results (Table 5.8).

5.2.4 Web Browser - Google Chrome

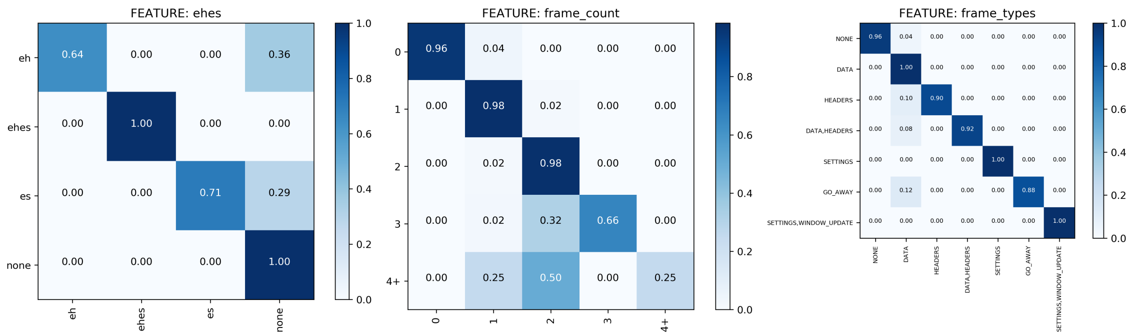

Figure 5.10: Confusion matrices for the "browser-chrome" experiment: cross-validation results for "ehes", "frame_count" and "frame_type" classifiers

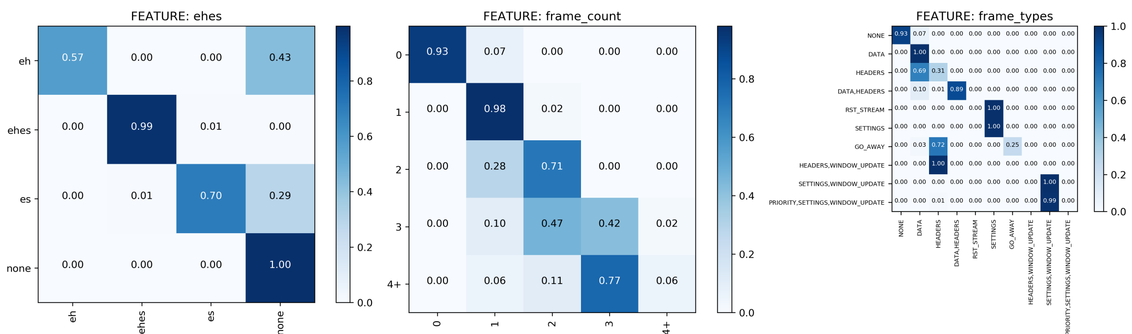

Figure 5.11: Confusion matrices for the "browser-chrome" experiment: prediction results for the "ehes", "frame_count" and "frame_type" classifiers using samples from "browser-firefox".

	Chrome *versus* Chrome			**Chrome *versus* Firefox**		
	ehes	frame_count	frame_types	ehes	frame_count	frame_types
Correct	4001	3966	4023	19228	17934	18236
Incorrect	52	87	30	300	1594	1292
Accuracy	98.7%	97.9%	99.3%	98.5%	91.8%	93.4%

Table 5.10: Number of correctly and incorrectly classified samples and classification accuracy from the cross-validation results and prediction results using samples from "browser-firefox".

5.2.5 Web Browser - Mozilla Firefox

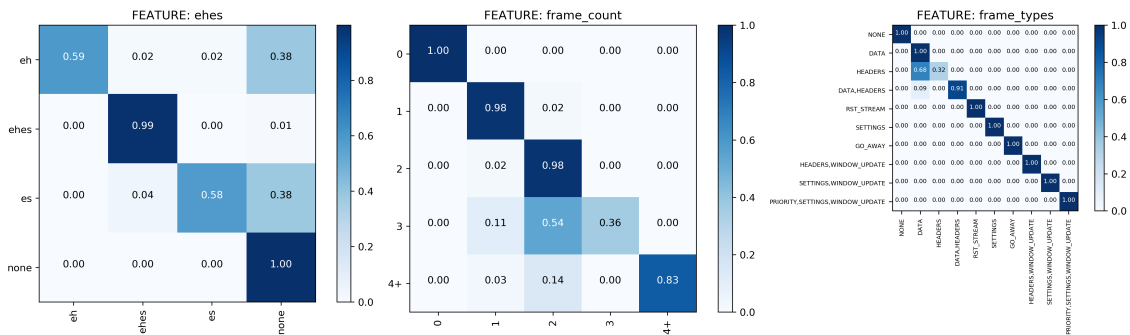

Figure 5.12: Confusion matrices for the "browser-firefox" experiment: cross-validation results for "ehes", "frame_count" and "frame_type" classifiers

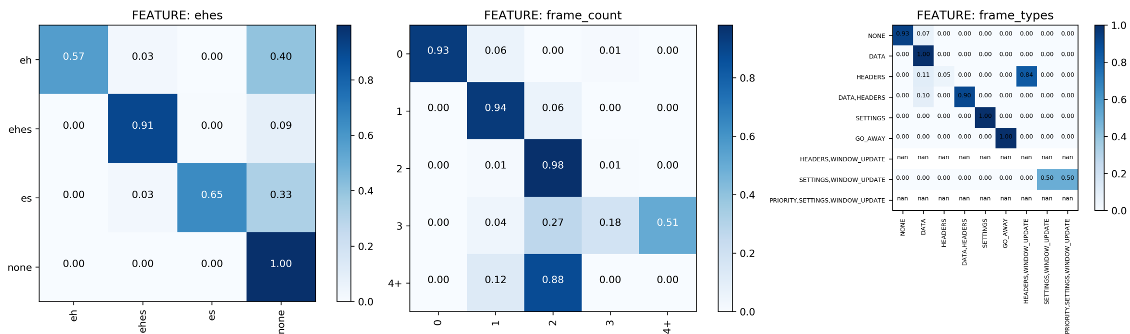

Figure 5.13: Confusion matrices for the "browser-firefox" experiment: prediction results for the "ehes", frame_count and "frame_type" classifiers using samples from "browser-chrome".

	Firefox *versus* Firefox			**Firefox *versus* Chrome**		
	ehes	frame_count	frame_types	ehes	frame_count	frame_types
Correct	3843	3801	3872	19868	18971	19328
Incorrect	63	105	34	393	1290	933
Accuracy	98.4%	97.3%	99.1%	98.1%	93.6%	95.4%

Table 5.11: Number of correctly and incorrectly classified samples and classification accuracy from the cross-validation results and prediction results using samples from "browser-chrome".

5.3 Discussion

The differences in the HTTP/2 protocol implementation among web browsers and web engines are noticeable in the results of the machine learning experiments presented in this section of the report. The different versions of the same software product also bring minor changes that were

evident, despite interfering with the classification results to a lesser extent. The lack of diversity among samples and captures could be improved by performing real-world website captures.

The "ehes" and "frame_types" classifiers provided reasonable predictions with minor deviations, even when attempting to classify samples from different web browsers and engines. The results "frame_count" classifier were not sufficient, since cross-validation results shows perceptible classification errors for TLS records having more than two HTTP/2 frames. Reducing the number of classes and grouping records with four or more frames into a single class to avoid class imbalance further improved accuracy by a few points.

In terms of extensibility, the architecture of the machine learning component is modular, meaning that adding a new classification algorithm or fine tuning the parameters of existing algorithms is trivial and requires negligible changes to the source code. A possible improvement for the training tool would be automatic parameter tuning. A similar approach to what the tool does in order to choose the best algorithm using cross validation could be applied to choosing the best parameters for that particular algorithm and classifier. The machine learning component can be further extended with new classifiers to enrich the quantity and quality of the predictions made, only requiring a "translation" function to convert such information to their equivalent classes.

Chapter 6

Conclusions

This dissertation addressed a well-known Internet privacy problem regarding encrypted traffic analysis and website fingerprinting. Nevertheless, there was an attempt to innovate with a different approach to this problem, using modern technologies such as the HTTP/2 application-layer protocol and web browsers with a mature implementation of the full HTTP/2 protocol specification. This dissertation attempted to combine subjects such as information security and privacy of users on the Internet with secure web application development, programming standards and good practices in computer engineering. Internet privacy and network security are both areas of great research interest, mainly due to the ethical and social implications associated with the technological advancements observed in the millennium. It is also expected that in the future there will be a greater research effort on web traffic fingerprinting.

The capture process and web browser interaction automation brought some degree of automation that ensured minimal user interaction and replication of this procedure on multiple machines with the same outcomes. The charts provided by the visualization component helped identify unique features of HTTP/2 client-server communication and HTTP/2 frame characteristics, such as their length distribution and count by type. The differences in the HTTP/2 protocol implementation among web browsers and web engines were noticeable in the results of the machine learning experiments, while newer releases of the same software product (for instance, *NGINX* 1.10.3 *versus NGINX* 1.14.0) bring minor changes and interfere with the results to a lesser extent.

The developed solution is modular and extensible. If there is need to support more protocols in the future, the capture automation tool does not require further intervention, as long as the traffic is generated over TCP or UDP protocols on a network interface. Less common protocols and devices would require a packet capture tool other than *tcpdump*. The only limitation of the parser component is that it relies on the PDML output generated by *tshark* for reading packet information, thus the protocol being analyzed must also be supported by the tool. Nevertheless, all three components of the analysis tool can be extended to support HTTP/1.1 over TLS or even the new QUIC protocol with little effort.

6.1 Future work

Three possible developments were identified for this project: the first and most relevant consists of implementing complex and high-level machine learning classifiers capable of identifying web object content types, the pages visited by the users or even inferring about user interaction within a particular website and recognizing patterns such as the sequence of pages visited, if this particular user is logged in or visiting as a guest, was visiting the website as a guest and registered a new account, purchased or searched for products in the catalog page (for instance, in an online store). As mentioned throughout this report, these machine learning algorithms should rely only on features provided by the encrypted network traffic, mostly metadata extracted from application layer payloads or derived from packet flow statistics.

The second development consists of surveying a list of popular websites (from a well-known and reputable source, such as *Alexa* Top 500 Global Sites) from different markets (banking, news, health care institutions) and generate web traffic by simulating typical user interactions on these websites. Execute the machine learning algorithms developed for frame and web object classification against the generated traffic in order to assess whether each market is more or less vulnerable to fingerprinting and evaluate the variance inside each market.

Lastly, the analysis tool components could be extended to perform similar feature extraction on QUIC network traffic. Quick Internet UDP Connections (pronounced "quick") [HISW16] is an experimental transport layer protocol developed by Google that implements HTTP/2 multiplexed connections between two endpoints over the User Datagram Protocol (UDP), while assuring reliability, manageability and fault tolerance. It was designed to provide security features equivalent to TLS/SSL over TCP connections, bandwidth estimation for congestion control on each direction, along with reduced connection and transport latencies typically associated with UDP connections. The main goal of QUIC is to improve performance of connection-oriented applications that currently use TCP as their primary transport protocol. QUIC also diverts control of the congestion avoidance algorithms into the user space at both endpoints, rather than kernel space, which should allow the efficiency of these algorithms to improve. The Internet Draft specification for this protocol was submitted to the Internet Engineering Task Force (IETF) for standardization in June 2015, and a working group was established in 2016. Statistics suggest that QUIC is currently supported by approximately 0.8 percent of web servers.

References

[BPT15] M. Belshe, R. Peon, and M. Thomson. HyperText Transfer Protocol Version 2 (HTTP/2). RFC 7540, May 2015. https://tools.ietf.org/html/rfc7540.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Mach. Learn.*, 20(3):273–297, September 1995.

[CWWZ10] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow, booktitle = Proceedings of the 2010 IEEE Symposium on Security and Privacy. SP '10, pages 191–206. IEEE Computer Society, 2010.

[DDPH17] R. Dubin, A. Dvir, O. Pele, and O. Hadar. I Know What You Saw Last Minute: Encrypted HTTP Adaptive Video Streaming Title Classification. *IEEE Transactions on Information Forensics and Security*, 12(12), December 2017.

[Fel10] Ed Felten. Side-Channel Leaks in Web Applications, March 2010. https://freedom-to-tinker.com/2010/03/23/side-channel-leaks-web-applications.

[Fou15] Mozilla Foundation. Firefox Release Notes (36.0), February 2015. https://www.mozilla.org/en-US/firefox/36.0/releasenotes.

[HISW16] Ryan Hamilton, Jana Iyengar, Ian Swett, and Alyssa Wilk. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2, January 2016. https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02.

[kn:18a] Matplotlib: Python Plotting, June 2018. https://matplotlib.org.

[kn:18b] NetworkX, June 2018. https://networkx.github.io.

[kn:18c] scikit-learn 0.19.1 documentation, June 2018. http://scikit-learn.org/stable/index.html.

[Lab09] Qualys SSL Labs. HTTP Client Fingerprinting Using SSL Handshake Analysis, June 2009. https://www.ssllabs.com/projects/client-fingerprinting/index.html.

[Mor17] Ricardo Morla. Effect of Pipelining and Multiplexing in Estimating HTTP/2 Web Object Sizes. *CoRR*, abs/1707.00641, July 2017.

[Net18] Netcraft. January 2018 Web Server Survey, January 2018. https://news.netcraft.com/archives/2018/01/19/january-2018-web-server-survey.html.

[Pro18] The Tor Project. Overview, February 2018. https://www.torproject.org/about/overview.html.en.

[SRBS17] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. The Onions Have Eyes: A Comprehensive Structure and Privacy Analysis of Tor Hidden Services. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 1251–1260, Republic and Canton of Geneva, Switzerland, April 2017. International World Wide Web Conferences Steering Committee.

[Ven15] Venturebeat. Google adds full HTTP/2 support to latest Chrome build, March 2015. https://venturebeat.com/2015/03/17/google-adds-full-http2-support-to-latest-chrome-build.

[W3T18] W3Techs. Usage Statistics and Market Share of Web Servers for Websites, February 2018, February 2018. https://w3techs.com/technologies/overview/web_server/all.

[WBC+08] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson. Spot Me If You Can: Uncovering Spoken Phrases in Encrypted VoIP Conversations. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 35–49, May 2008.