
VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
DEPARTMENT OF COMPUTATIONAL AND DATA MODELING

Master's thesis

# Deep packet inspection algorithms and applications

Done by:
Rimantas Stankevičius signature

Supervisor:
Eduardas Kutka

Vilnius
2021

# Contents

# Definitions

- **DPI** - *Deep Packet Inspection* - a network packet inspection method that includes packet payload information in the analysis.
- **TCP** - *Transmission Control Protocol* - connection-oriented transport layer protocol with reliability features.
- **UDP** - *User Datagram Protocol* - connectionless transport layer protocol.
- **QUIC** - transport protocol that is based around UDP with TCP-like features, and is usually implemented in application layer, unlike TCP. Name is not an acronym.
- **HTTP** - *HyperText Transfer Protocol* - application-layer protocol designed for hypermedia information systems.
- **HTTPS** - *HyperText Transfer Protocol Secure* - a secure variant of HTTP protocol that provides confidentiality, integrity, authenticity guarantees.
- **HTTP/1.x** - in this paper this refers to first major HTTP version and its sub-versions HTTP/1.1 and HTTP/1.1.
- **HTTP/2** - HTTP protocol major version 2, published as IETF RFC in 2015, binary protocol unlike HTTP/1.x, and more feature-rich.
- **HTTP/3** - HTTP protocol major version 3, not yet finalized. Evolution of HTTP/2, uses QUIC protocol as transport.
- **FSM** - *Finite State Machine* - computational model of state machine with fixed number of states.
- **DFA** - *Deterministic Finite State Automaton* - A class of FSM where there is only one possible transition between states for given input.
- **NFA** - *Non-deterministic Finite State Automaton* - a class of FSM where multiple transitions to the next state are possible for given input.
- **DNS** - *Domain Name System* - a protocol for translating human-readable resource names into IP addresses.
- **TLS** - *Transport Layer Security* - a protocol for allowing secure communications. Commonly used with HTTP protocols.
- **DoH** - *DNS over HTTPS* - a protocol for performing DNS queries over HTTPS.
- **VPN** - *Virtual Private Network* - a technology for extending and connecting to a private network over internet.
- **ALPN** - *Application Layer Protocol Negotiation* - TLS extension that allows applications to negotiate which protocol they will use for communication.
- **packet** - in this paper the term refers both to TCP segments and UDP datagrams.

# Abstract

Deep packet inspection is a technology that can be used for reasons varying from local network protection and monitoring to internet censorship. The algorithms that can be applied are actively researched, and mostly are focused on the improvement of the data structures that can be employed. As the percentage of encrypted internet traffic increases, and some algorithms become less usable. There is a need for other methods of packet analysis that do not disrupt encryption.

One of the methods is web traffic fingerprinting. It allows websites and mobile apps to be identified from their traffic patterns. This idea has been experimented with various protocols, including HTTP/1.1 and HTTP/2. This work tries web traffic fingerprinting with HTTP/3, newest HTTP protocol which is yet to be finalized.

In result, it was shown that HTTP/3 web traffic can be fingerprinted with similar accuracy as traffic of other HTTP protocols, when considering some aspects of the protocol. It was found that it must be taken into consideration that HTTP/3 traffic is more encrypted and some things as acknowledgement packets must be detected statistically in order to increase accuracy of fingerprinting methods.

# Santrauka


**Giluminio paketų tikrinimo algoritmai ir jų taikymas**

Giluminis paketų tikrinimas yra technologija kurią galima naudoti įviariais tikslais, pradedant nuo vietinio tinklo apsaugos ir stebėjimo, iki interneto cenzūros. Tiems tikslams naudojami algoritmai yra aktyviai tiriami, ieškant kaip juos paspartinti, ar optimizuoti atminties naudojimą. Dėl didėjančio užšifruotų interneto komunikacijų procento kai kurie naudojami algoritmai tampa mažiau panaudojami. Atsiranda poreikis analizuoti tinklo paketus kitais būdais, nenutraukiant šifravimo.

Vienas iš metodų yra tinklo srauto atspaudavimas. Tai leidžia identifikuoti mobliasias programėles ir internetinius puslapius iš jų sukuriamų tinklo srauto šablonų. Ši ideja buvo išmėginta su įvairiais protokolais, įskaitant HTTP/1.1 ir HTTP/2. Šiame darbe tinklo srauto atspadavimas išbandomas su HTTP/3 protokolu, naujausiu HTTP protokolu, kuris dar nėra užbaigtas.

Rezultatai parodė, kad HTTP/3 puslapių tinklo srautas gali būti atspauduojamas panašiu tikslumu kaip ir kitų HTTP protokolų srautas, atsižvelgiant į kai kuriuos protokolo aspektus. Buvo rasta, kad reikia atsižvelgti, jog HTTP/3 protokolu didesnė srauto dalis yra užšifruota, ir kai kurie dalykai, tokie kaip patvirtinimo paketai turi būti aptikti statistiškai, tam kad padidinti atspaudavimo metodų tikslumą.

# Introduction

Deep packet inspection has many different applications, ranging from intrusion detection systems parental controls, quality-of-service control, and even internet censorship. Deep packet inspection technologies evolve with the underlying network technology - protocols, hardware and user demands. Ideally the technology should be able to inspect all traffic and make decisions in real time. This leads to developing new algorithms and optimizing existing ones to match ever-increasing network bandwidth.

One particular problems that faces DPI is the spread of encryption. HTTPS, while providing confidentiality, integrity and authenticity, in most cases makes it impossible to perform deep packet inspection. Although breaking end-to-end connection sometimes can be possible without causing too many issues to the user, it could be argued that the method defeats the whole purpose of HTTPS.

There are relatively recent protocols, such as BlindBox[18] and PrivDPI[16], that provide a way to perform deep packet inspection over encrypted data. While providing privacy guarantees to the user, and allowing other parties to do deep packet inspection, they have not spread in any real-world applications. Moreover, they have other problems, such as more complicated setup requirements than HTTPS, and lower throughput due to additional bandwidth and computational requirements. Considering the advantages, the mentioned downsides are not critical and are likely to be reduced with more research that would come with more widespread usage of protocols. There are no implementations of BlindBox or PrivDPI that are widely used. At the moment they are a topic of academic research and not of a real-world applications.

Another way to analyze encrypted web traffic is to analyze packet metadata, such as timings, packets sizes, and other derivable values. This approach does not reveal actual data sent but can provide useful data to system administrators. Examples of this type of data would be mobile applications, or websites being browsed. This is especially useful when keeping in mind advancements in technologies such as DNS-over-HTTPS[10] and encrypted server name identification[17].

It has been shown by multiple papers that it is possible to analyze encrypted web traffic and with high degree of confidence fingerprint what website has been visited by a user. One particular aspect that must be kept in mind, is that there are multiple protocols used on the web, specifically multiple versions of HTTP protocols (HTTP/1.x, HTTP/2, upcoming HTTP/3). Due to differences between protocols and different features supported by them, network traffic patterns created by each protocol are expected to differ at least to some degree. Currently there are research papers for HTTP/1.x and HTTP/2. Since there is HTTP/3 protocol, although still not yet completed, it is worth looking into it, and checking if there are any obstacles that might reduce success of fingerprinting websites from their traffic.

The main goal of this paper is to check whether usage of HTTP/3 protocol causes worse results in website fingerprinting, and if it does, check for possible improvements. This paper will research all three major HTTP protocol version results, with HTTP/1.1 and HTTP/2 providing baseline results that can be verified by comparing results form other papers, and use them to compare against HTTP/3. Since HTTP/3 is not finalized yet, it is expected that there will be minor changes to the protocol but it is unlikely that they will be significant enough to alter results of this paper in a major way.

In result this paper will complement existing research by giving comparison on how different HTTP versions affect difficulty of fingerprinting.

Small part of this paper is taken from last semester's work. This is mainly literature overview

of overall subjects and algorithms in deep packet inspection. Additionally, some descriptions of last semester's work is included, and a few drawing are taken; all other research and writing has been done this semester.

The paper consists of three main sections. First section is an overview of deep packet inspection. Second section consists of more detailed overview of web traffic fingerprinting at different network stack layers. Main focus are HTTP protocols, including their features and fingerprinting methods. Third section describes practical work. This includes experimental setup, data collection, and initial results followed by adjustments.

# 1 Topic Overview

## 1.1 Packet inspection methods

Network packet inspection can be grouped depending on how much of the data is inspected from the data stream. For example, shallow, or stateless, packet inspection deals only with packet headers, and is limited to information such as IP addresses, ports, and monitoring unusual traffic flows. A simplest form of packet inspection can be an access control list in a firewall or other type of network equipment that defines rules whether a packet can be allowed through firewall. Parameters include but are not limited to source and destination IP addresses, ports, transport layer protocol, application protocols. The advantage of such approach is that it requires very little computational power. As as a consequence of that, it is fast, as the data fields that need to be validated usually are strictly defined in packet headers and are usually small. Port numbers, IP addresses usually fit in one machine word each. However, this method of packet inspection has limited capabilities. There is a need for more thorough packet inspection.

Deep packet inspection (DPI) techniques inspect not only packet headers but also packet payload. For example, deep packet inspection software can inspect HTML content for malicious scripts, or scan file transfers for malware, detect content type even though it is not being transferred on an expected port. Deep packet inspection goes beyond data of single packets and can use accumulated information in order to make decisions about the network traffic and how it should be dealt with. This analysis can encompass the whole network communications. Some of the examples where deep packet inspection is used are:

- Intrusion detection systems which monitor network traffic for any malicious payloads.
- Protocol analysis. Known protocols can be monitored for any deviance from the norm, which could indicate some sort of attack.
- Network analysis.
- Throttling network speeds for specific content types.

## 1.2 Common Algorithms

Two important aspects in the DPI research are speed and complexity. Since DPI deals with packet payloads, it has to be able to identify and process many different payload types and process data quickly, especially if it must be done in real time. It should be noted that real-time is not a requirement for DPI as there might be applications where data is collected first and analyzed later.

A relatively simple deep packet inspection method is pattern matching. It can be a simple string matching but usually regular expressions are used for their capabilities. Regular expressions can be implemented, or rather compiled to, a finite state machine (FSM). In short packet contents are fed into a FSM equivalent of a regular expression, and if the FSM stops in the accepting state, then the pattern is matched. Depending on computational and memory constraints, a deterministic finite state automata (DFA) or non-deterministic finite state automate (NFA) can be used. As an example, non-deterministic finite state automata (NFA) is has greater computational complexity because every path it can take must be computed. In DFA, there is only path for state transitions but the DFA usually has more states and transitions more states than equivalent NFA.

Since both types of finite state automata have some drawbacks, there are research efforts in their optimization. For instance, in [14] Kumar et al. created a method for reducing complexity of the FSM by reducing the number of states, and gives basis for a hardware-based implementation. The proposed method reduces number of states in given FSM by introducing a default states, that is, a states that are transitioned to when there are no other valid stat transitions for given input. The authors show that they were able to reduce number of state transitions by more than 95% for given data sets. [19] shows that NFAs parallelization can be optimized for a specific hardware architecture, while [14] [22] shows algorithms for optimizing regular expressions, targeting patters that commonly appear in deep packet inspection applications, and providing rules for rewriting regular expressions to be less computationally expensive.

There are other methods besides regular expressions. For instance, use of probabilistic data structures, such as bloom filters. Bloom filters are efficient data structures that are able to check whether an element is a set, with the property that adding an element to the set does not increase the size of the data structure. Checking whether an element exists in the bloom filter can result in false positives but never in false negatives. Another important property of bloom filters is that the time needed to add or check whether an element in a set is constant. Desired false positives rate can set during initialization of the data structure.

Dharmapurikar et al.[6] propose a method of using parallel bloom filters combined with a deterministic algorithm. First, a bloom filters are used to find possible matches, and if match found, the match is further analyzed in order to prevent false positives. Also, bloom filters can have very efficient hardware implementations because application-specific integrated circuits for hashing functions exist, and operations required to check whether an element is likely in the set, can be parallelized. Boom filters are not the only probabilistic data structure used and researched for DPI, [4] shows that quotient filter can be faster between 30% and 75% than a bloom filter. Cuckoo filter can also be better performing than a bloom filter[5]. The problem with bloom filters is that they cannot be used to tell which element has been detected but for some purposes it might be sufficient. Also, once an element is added, it cannot be removed except by rebuilding the data structure. This can be solved by using a counting bloom filter.

As for practical implementations of deep packet inspection, although the algorithms can be run on a general purpose computer hardware, many authors in their papers include hardware-based implementations. It is often noted that they are less flexible than pure software implementations, when implemented correctly they offer greater speeds. It should be noted that graphics processor units are in many cases well suited for DPI tasks due to parallelization. In other cases, special hardware can be used, for example, ternary content addressable memory for implementing regular expressions.

## 1.3 Problems With Encryption

One of the main challenges deep packet inspection faces is ever-increasing encryption. According some sources [1] about 82 percent of web traffic is encrypted. The percentage has no reason to decrease and it is understandable because with unencrypted traffic all components of CIA triad (Confidentiality, Integrity, Availability) can be breached. HTTPS offers integrity, authenticity, and confidentiality, depending on the implementation. However, deep packet inspection is either not possible or has to compromise security to some degree in order to fully analyze network traffic.

[1] https://blogs.cisco.com/security/threats-in-encrypted-traffic

For example, one of the methods for analyzing HTTPS traffic is to break end-to-end connection between client and desired server. This method can be applied in corporate networks, for example, by pre-installing a root certificate of the company on the employees computers. "Mitmproxy"[1] is an example of proxy software that does exactly that. It generates a certificate which can be installed on user's computer and then "Mitmproxy" can intercept HTTPS requests. Improper handling and usage of such tools, however, heavily compromises security, as it happened with Lenovo's "Superfish"[2] adware, which came pre-installed in users' laptops. The ethics of adware are out-of-scope but the relevant part is that it even contained certificate's private key that in hands of the attacker could be used to generate fake certificates and execute man-in-the-middle attacks.

Even though that percentage of encrypted traffic is increasing but not all newly-encrypted protocols are considered without problems. DNS, originally being a plaintext protocol, has multiple extensions which provide various levels of security. The most recent protocol is called DNS-over-HTTPS (DoH). It allows performing DNS queries over HTTPS protocol, which makes it very hard to detect that DNS query is even happening. Another problem that has been criticized is that there is a small number of trusted DoH servers, which might be used to collect users data.

It is also a problem for law enforcement, and there have been attempts to weaken encryption, even before widespread use of encryption internet technologies, such as clipper chips that were supposed to allow government organizations to intercept voice or text messages by leaving a backdoor.

There are two different approaches for performing DPI over encrypted traffic. First, analysis of metadata (parameters of TCP/IP stack, duration, timings, payload sizes) and development of new protocols that allow performing deep packet inspection while providing privacy guarantees.

## 1.4 BlindBox and Related Protocols

BlindBox by Sherry et al.[18] is a proposed first of its kind network protocol that allows to perform data inspection over encrypted traffic. This is achieved by having a secondary stream created from encrypted plaintext tokens that can be analysed by network middlebox without decrypting them.

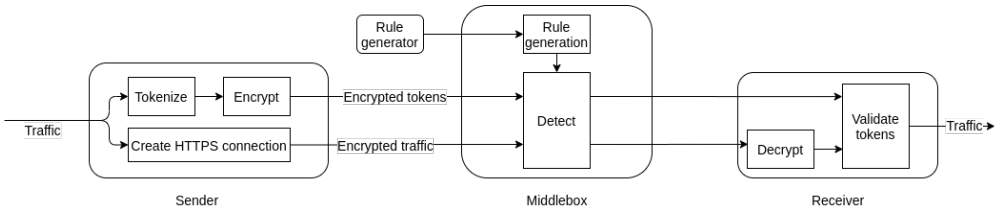


Figure 1. Protocol schema based on BlindBox and PrivDPI papers.

The protocol schema is shown in fig. 1. It can be considered an extension of HTTPS or TLS protocol, as the connection is established without breaking HTTPS end-to-end encryption, and then establishing a secondary connection which encrypts the same traffic sent via HTTPS connection, but in a form that allows middlebox to scan the data without decrypting it. This is achieved by splitting payload into fixed size tokens, encrypting them and sending to the middlebox. Middlebox also splits IDS rules it wants to check into fixed size tokens and encrypts them. Having payload and IDS rules encrypted with the same key allows comparing them. The protocol provides algorithms for encrypting payload and IDS rules with the same key without revealing IDS rules to the sender/receiver and without revealing the encryption key to the middlebox. Middlebox without knowing the encryption key cannot decrypt payload and only compare it to IDS rule tokens. At the

same time sender/receiver does not know IDS rules, which it might try avoid, or that they might be a commercial secret.

As noted in paper titled PrivDPI[16], which builds on BlindBox, this protocol would allow easier compliance with data protection laws, as GDPR (General Data Protection Regulation) because it could be configured to avoid examining personal data. Although the protocol can also work in "probable cause privacy" mode, which allows full decryption of the data if the encrypted data contains a suspicious string. Unlike traditional man-in-the-middle solutions which try to be transparent between the client and the server, Blindbox and PrivDPI require both parties to be aware of the network middlebox. It should be noted that BlindBox is the first protocol of its kind, for practical applications protocol outlined in PrivDPI[16] is more suitable.

The protocol is not without its drawbacks. The first problem with the proposed protocol is that it requires all parties to support the protocol - client, server, and the middlebox that performs the packet analysis, so it cannot be easily integrated into existing infrastructure. Having two connections for each request also makes implementation more complicated. Another drawback is traffic overhead caused by the encrypted tokens stream, which based on the original protocol requires 5 times the original payload. This means that 1Mb transfer via HTTPS is  6Mb via PRivDPI (5x overhead plus HTTPS payload).

The first issue might be partially solved by making it optional and switching to it via application layer protocol negotiation (ALPN). It might be possible to avoid having two separate connections by combining them under one protocol. An example idea for this could be using QUIC protocol and designing a stream that is readable by a middlebox. Substituting middlebox with a proxy, it should be possible to design a system that employs a single connection.

Even with discussed drawbacks, both BlindBox and PrivDPI are more suited for real-world use than other encryption schemes, such as homomorphic encryption.

## 1.5 Metadata-based Deep Packet Inspection

Encrypted communications still leak some metadata that can be used to determine some information about encrypted traffic. One source of metadata is flow data, such as packet sizes, timings, which can be used for making content type predictions based on statistics. TLS handshake information also provides valuable information which can be used to determine information about client and server software, which in turn can be combined into a fingerprint, for example, JA3 hash[2]. This data can be used to detect encrypted malware communications as outlined in in paper by Anderson et al.[11], or for other use cases such as detection of bots due to TLS differences between browsers and software libraries, such as curl. Handshake message sequences can be analysed to guess the target web page being requested, as shown in paper "Markov Chain Fingerprinting to Classify Encrypted Traffic" by Korczynski et al[13]. Although their method only analysed a small subset of of websites Even TCP stack parameters can be used to identify information about hosts, such as operating system. This is not limited just to operating system or browser fingerprinting. Observing encrypted connection metadata can be used to predict websites being browsed, or in case of smartphones, mobile apps being used, as demonstrated by Hintz[9] and Tylor et al.[20] This is an actively researched topic due to the reason that it is more practical than new protocols that provide deep packet inspection capabilities and privacy protections.

In some specific cases encrypted communications are not safe from inspection. One such case

[2]https://ja3er.com/

has been shown in a paper "Uncovering spoken phrases in encrypted VoIP conversations"[21]. Researchers exploited the fact that VoIP codec produces different packet lengths for different phonemes and were able to recover from 50% to 90% of phrases. This, however, should not be considered a weakness of encryption itself but rather implementation detail and the issue was fixed by padding the data packets.

# 2 Metadada-based Traffic Analysis

## 2.1 Motivation

Normally, in order to inspect HTTPS traffic, a network middlebox would perform what is know as "HTTPS inspection" (fig. 2, 3). The network middlebox acts as a man-in-the-middle, breaking end-to end connection from the client to the server. This action should cause the connection to be aborted by any sensible web browser or any library that supports HTTPS. In order to prevent that, the network administrator would pre-install root certificate of the middlebox on the users' machines.

The HTTPS inspection might not be desirable, since it breaks end-to-end encryption and defeats the whole purpose of the HTTPS. Depending on the jurisdiction the organization which performs HTTPS inspection might have difficulties implementing it due to data protection laws. For instance, users might be sending personal or other sensitive information such as credit card numbers of bank account credentials.

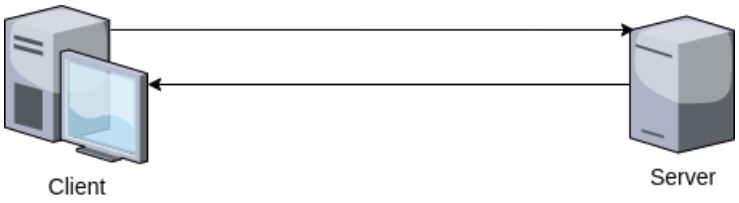


Figure 2. Normal HTTP/HTTPS communication.

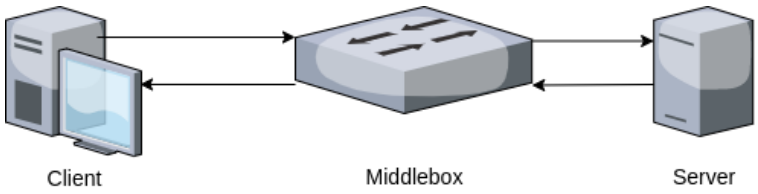


Figure 3. Connection with HTTPS inspection. End-to-end connection is broken in the middlebox.

There are new protocols which provide a balance between security and privacy, as mentioned in previous section, but this paper will focus on HTTPS, and each individual HTTP version (HTTP/1.1, HTTP/2, and HTTP/3). Since deep packet inspection is impossible over encrypted traffic, we can still collect metadata and use information from unencrypted packets. Based on collected data, it is possible to guess operating system, browser or other type of application that made the requests. This is possible due to the fact that different systems can have different default values of protocol fields, have different capabilities, or respond differently to various sequences of packets. Following sections will focus on transport and application layers of TCP/IP stack but it should be noted that other layers can be exploited, as far as physical layer, allowing to determine device that is sending data by signal analysis[7].

## 2.2 Threat Model

In our threat model an attacker develops fingerprints of a set of web pages that use HTTPS protocol, and wants to be able to identify them in a web traffic of targeted users. An attacker is able to capture web traffic in a network, and subsequently transport layer of TCP/IP model, which means

that information about source and destination IP addresses can be collected, and information from transport layer protocols, focusing on TCP and UDP. An example of where this kind of network traffic capture is possible would be a public WiFi spot controlled by an attacker, or even a hijacked ethernet cable. An assumption about targeted users is made that they do not use technologies such as virtual private networks, or anonymity services like TOR. Exact IP addresses will not be used in creating fingerprints of web pages because they have two problems. The first one is that a single web server might server multiple of websites, and second, there is no guarantee that a web page will be hosted on the same web page. Load balancing is also a factor. Also we do not rely nor try to use communications with DNS servers. DNS traffic is not expected to be always available due to caching, and with spread of usage of DoH protocol it becomes unavailable since it is encrypted. Lastly, even though we assume that we can observe individual destination addresses due to lack of VPN or TOR, avoiding usage of IP addresses makes it easier to apply gathered knowledge to cases when those protocols are used.

The goal of the attacker is to with a high degree of confidence find what websites have been visited by a user.

## 2.3 TCP Fingerprinting

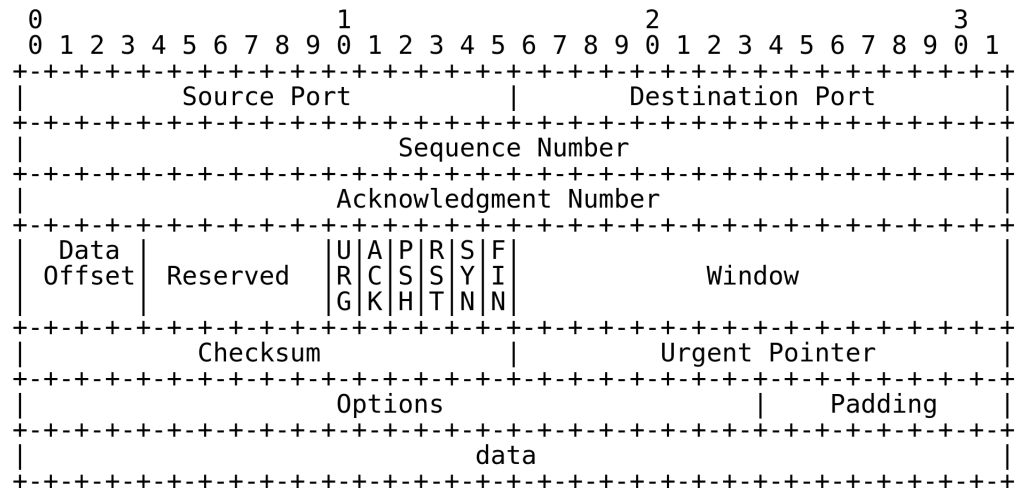


Figure 4. TCP packet structure. Source: https://tools.ietf.org/html/rfc793 section 3.1

Given a TCP packet structure (fig. 4) some fields are no use for fingerprinting, as they are specific connection parameters (eg. source, destination ports), or take predetermined values not influenced by the host configuration (checksum). What we are interested in are variables, such as window size, flags, options. Note that TCP packet size can vary depending on the options specified, so even packet size itself is a value that can be used for classification. It is possible to create a 67-bit fingerprint for a packet[3]. Tools like p0f usually focus on specific values and/or relations between values instead of using fingerprints.

Overall, TCP fingerprinting is best suited for operating system detection because with network stack usually being part of OS kernel, applications have limited influence over this layer. TCP is relatively stable which limits possible new ways to fingerprint it.

## 2.4 TLS Fingerprinting

Protocols which are used in application level and less hardware-dependent are more prone to change, thus increasing their footprint of fingerprinting possibilities. TLS is a good example in this category because it has multiple versions, multiple extensions, and other unencrypted information.

TLS (Transport Layer Security) protocol is a protocol which provides confidentiality and integrity for transport layer communications (TLS for TCP, DTLS for UDP), with optional authenticity. Authenticity is optional and usually is one-way, that is, client authenticates server but the server does not authenticate client. Communication over TLS protocol can be divided int two stages - protocol negotiation (handshake) and application data transmission. Application data is encrypted, and does not provided much information besides its size but protocol negotiation is unencrypted and provides valuable data that can be used to determine information about application and operating system. An example where this information can be applied is detection of web scrapers. By collecting fingerprints of popular browsers versus software tools such as curl, or other programming language-specific libraries, a network observer or a website owner can apply countermeasures to prevent activity of web crawlers. Most easily identified data in TLS fingerprinting are TLS version, cipher suites, extensions (eg. ALPN).

In the end, both TLS fingerprinting may provide information about software of the communicating parties but information about what is actually being requested is limited. And although it is possible to find host name in TLS server name identification extension, it will be less possible if encrypted server name identification feature will become widespread. That is likely because it provides additional confidentiality, leaving TLS handshake information less valuable.

## 2.5 Fingerprinting over Encrypted Data

In this section we focus on assume that TLS protocols have no vulnerabilities that might reveal encrypted data, nor we attempt to perform man-in-the-middle attack. What else is left? Information extracted from TLS handshake, and encrypted packet metadata such as packet sizes and timings between packets. Now let us connect this data with what happens in a browser when user visits a website. For example visiting https://mif.vu.lt in chrome browser, one can open network tab in developer tools, and see that there are more than 60 requests made. Visiting the same page multiple times reveals that the order of requests stays the same, with little variation between sizes of equivalent requests.

If we think of those requests as fingerprint of a page, then we can try to guess the website being visited by observing packet metadata, such as timings, sizes, even if the traffic itself is encrypted. This has already been suggested and done with various methods by Hintz[9], Taylor et al.[20], and others.

Some technical details of protocols are usually skipped but are important, as they might affect overall data set we are trying collect for analysis, and the effectiveness of classification. Moreover HTTP protocols constantly evolve which also affects fingerprinting, as shown by Lin et al. with HTTP/2 protocol, who demonstrated that a "Server push" feature decreased fingerprinting accuracy from 80% to 74% in their benchmarks[15].

### 2.5.1 Variables That Affect Fingerprinting

Exact website fingerprint may vary depending on the HTTP protocol version, protocol features used, or HTTP headers. Currently there are three main HTTP protocol versions: HTTP/1.x, HTTP/2 and soon-to-be-finished HTTP/3.

HTTP/1.0 and HTTP/1.1 do not differ much from the point of encrypted traffic. One significant difference is that in HTTP/1.1 connections remain open by default (fig. 5, while in HTTP/1.0 it is not the default behaviour (fig. 6). In both versions of protocol this can be controlled with `Connection` header. This is important because if connection is closed after each request, then it is trivial to identify sizes of individual requests. If connection is not closed between requests then multiple requests are made within the same TCP connection, and data flow must be taken into consideration. Analyzing how packets move back and forth between client and server may allow to identify sizes of individual requests. Next important HTTP header that affects observed response sizes is `Compression`. Different browsers might support different compression algorithms which will affect response size. Chosen compression algorithm depends on client and server, as client informs server compression algorithms it supports via `Accept-Encoding` header.

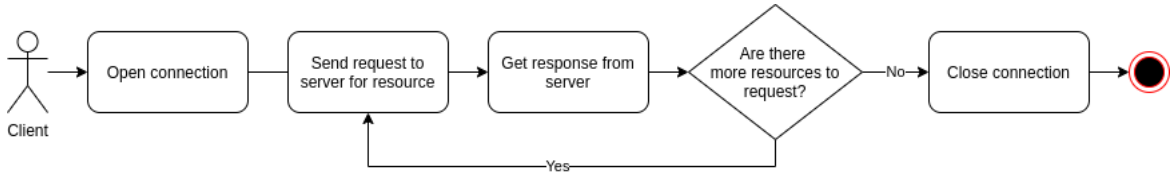


Figure 5. Multiple HTTP requests per connection, default for HTTP/1.1

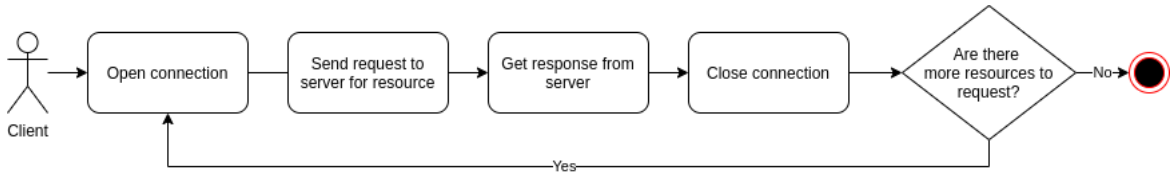


Figure 6. Single request per connection (HTTP/1.0 default, or HTTP/1.1 with `Connection: close`)

HTTP/2 differs significantly from HTTP1/1. First, it is a binary protocol, and it supports features that were not possible with older HTTP standards. For our analysis most important features are "server push" and multiplexing. Server push allows server to send data before it is requested by the browser (fug 7, stream 1). For example, if browser requests `/index.html` from server, and the server knows that browser is likely to requests `/style.css` because it is needed to properly render the page, the server can send that resource without the browser requesting it. The client indicates whether it supports this feature or not. If it is supported by both client and server, it makes harder to identify individual resource requests.

Next new feature is that requests can be multiplexed. In HTTP/1.1 multiple requests can be performed over a single connection but must be done sequentially. In HTTP/2 client can request multiple resources in parallel over a single connection. Each parallel request is called a stream, and sending requests over multiple streams is what allows multiplexing. This further complicates observation of individual resources being requested and sent.

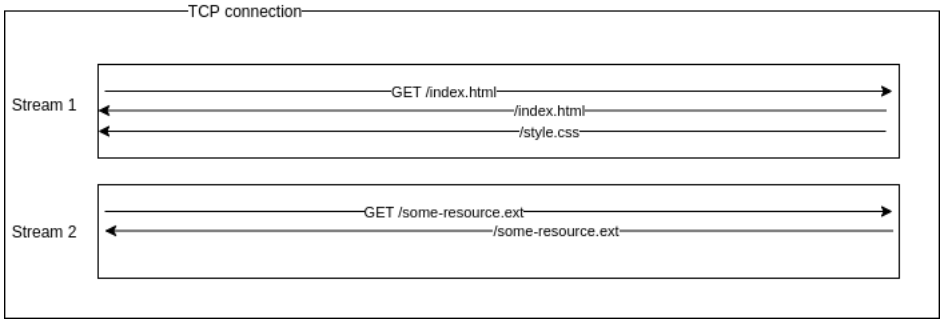


Figure 7. HTTP/2, streams, server push.

HTTP/3 shares many features with HTTP/2 with some differences. The major difference is that HTTP/3 does not use TCP as an underlying transport. Instead, it uses QUIC protocol, which in turn uses UDP. QUIC provides some functions that are available in TCP but not in UDP, such as congestion control and mechanisms dealing with packet loss. One of reasons to prefer QUIC with HTTP/3 over HTTP/2 and HTTP/1.1, is that QUIC solves head-of-line blocking problem. In HTTP/1.1 head-of-line blocking problem arises from the fact that requests over a single connection have to be performed sequentially. In HTTP/2 head-of-line blocking occurs because TCP layer is unaware of streams. For example, during a transmission of two streams a packet may be lost, and and that loss may block both streams until a packet is re-transmitted even though enough information has been transmitted for processing one stream. In HTTP/3 this is not an issue, as long as data for multiple streams is not in the same packet (QUIC IETF draft[12] section 13). Another feature of QUIC that arises due UDPs lack of concept of connections. If a connection is lost for example due to change of network (as in switch from WiFi to mobile network), there is no need to reestablish a connection. Each QUIC communication simply has an identifier that is used by client-server communication. This is in contrast to TCP, where connection would be lost and handshake process would be started again.

Second, there is no unencrypted version of HTTP/3, although it should be noted that HTTP/2 is very rarely used without encryption too. Other major HTTP/2 features are present, except for stream prioritization. Although HTTP/3 has limited support at the writing of this article, it must be taken into consideration, as otherwise a system trying to do fingerprinting might completely miss some traffic due its usage of UDP protocol.

Overall, the mentioned features of HTTP protocols make fingerprinting harder, even though there is no reason to believe this ever was the intention, as those features are mostly designed for performance improvements. Also it should be noted that advantages in ease of fingerprinting of HTTP/1.1 and HTTP/1.0 disappear if the traffic observed is going over VPN, as individual connections are not visible.

Lastly, web resources caching by browser has some limited affect on how the final website fingerprint will look like. If a user visits website for the first time, there will be more web traffic generated than for subsequent visits. This means that a website fingerprint will be more accurate if we generate fingerprint with browser caching.

### 2.5.2 Available Information

To fingerprint a website by its traffic we need to be able to identify and extract information from packet flow. We can distinct these main types of metadata available:

- Destination addresses.

- Packet flow direction.
- Timing.
- Amount of data transferred.
- Protocol (focusing transport layer).
- Individual connections, if supported by protocol.

Destination addresses are available as defined in threat model section. If all users connections are tunneled via VPN protocol, then we still have one destination address that of a VPN server.

Packet flow direction information is self-explanatory - any packet capture will have indication whether the packet is outgoing or incoming. Details that matter here are if the packet contains application data. Packets such as TCP ACK most likely should be ignored.

Timing information can be used for grouping series of packets by time intervals between them. This allows grouping of packets into bursts, which will be discussed later in this section.

Amount of data transferred will be used as the main data point. It is dependent on the information extraction methods but all information extraction methods will contains amount of data transferred, whether it is during a packet burst, or total data transferred to and from a destination address.

Transport layer protocol might have been irrelevant before QUIC and HTTP/3 protocols because earlier HTTP protocols work over TCP connections. HTTP/1.x is theoretically possible over UDP but not used. With HTTP/3 we must consider UDP, or otherwise web traffic that uses HTTP/3 will not be detected. This also brings observability of individual connections. Concept of connections does not apply to UDP since UDP is connection-less protocol.

Given these concepts we can define data tuples to be extracted from traffic capture. If we assume that we can observe individual connections at transport layer with only considering source and destination addresses, and packet direction, then the information of a single TCP connection $C$, results in tuple 2.1. Same can be applied to total information transferred between destination IP, disregarding individual connections to it.

$$C = (d, s, r) \tag{2.1}$$

where:

- $d$ - destination IP address. We assume that we observe a single source (client), so we can discard source IP address for further analysis. Destination IP can be used for counting how many connections are made per IP, including their order. TCP port information is also discarded for further processing, as for web traffic we are generally interested in port 443.
- $s$ - bytes sent. For example, based on the amount of bytes sent we can classify if the connection was used for making a file upload, or other type of request, such as HTTP GET. The assumption here is that file upload will generate significantly more bytes sent than a simple request.
- $r$ - bytes received. Signifies an approximate downloaded resource size.

The information above is sufficient in an ideal case, where HTTP version below 2 is used, and connection is closed between requests. As a starting example let us consider HTTP/1.0 and HTTP/1.1 with connection not being closed between requests to the same IP. Information in the form of tuple (2.1 will not distinct individual resources. For this case we can analyze data flow, which is rather simple, since HTTP/1.0 and HTTP/1.1 do not support multiplexing requests between connections. In that case when a client sends data to server, we can treat it as beginning of a request, which will be followed by server's response. After that, if client sends data to server, we can treat it as a beginning of a new request-response cycle. Each cycle of data transmission and reception can be called a flow (2.2). Resulting connection information is a tuple of destination IP address followed by flows (2.3). Flowchart of how flows are processed can be seen in fig. 8

$$F = (s, r) \tag{2.2}$$

$$C = (d, F_0, F_1, ...F_i) \tag{2.3}$$

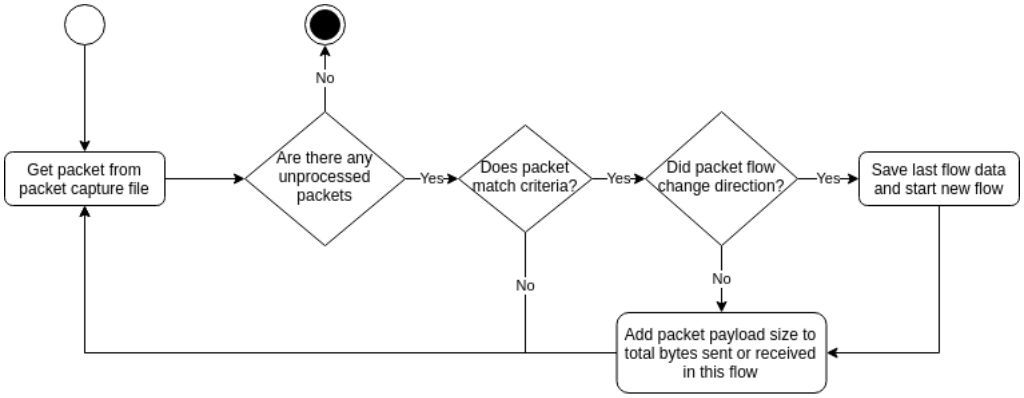


Figure 8. Extracting flow information from packet capture. This is done for each destination IP separately

Above tuples are also valid if we ignore connections and instead monitor overall traffic between destination and source IP addresses.

We can use timing information to analyze packet capture. Burst is a group of packets that are related by the fact that they occur in a a timing sequence where time between each packet is less than burst threshold (fig. 9). The idea of burst is taken from Appscanner by Taylor et. al.[20]. In that paper burst are split into flows, where each burst consists of one or multiple flows. Flow is defined as a sequence starting with outgoing packet, followed by incoming packets, and terminated by a following sequence of outgoing packets. We can devise four types of bursts, depending on whether they are destination-aware and if they include flows:

1. Destination address-aware burst without flow information.
2. Destination address-unaware burst without flow information.
3. Destination address-aware burst with flow information.
4. Destination address-unaware burst with flow information.

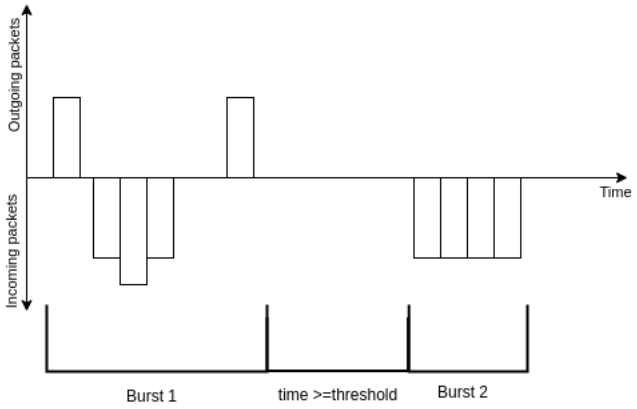


Figure 9. Packet burst example. New burst starts when there is inactivity longer than the burst threshold.

Unlike in Appscanner, in this paper we will not split bursts into flows and flows will be used independently of bursts as shown in fig. 10. Destination-aware means that burst will be calculated per each destination (server) IP address, while destination unaware burst is calculated form overall traffic that matches criteria, e.g TCP protocol port 443. Burst types set minimal number of bursts that can be extracted from a packet capture of a web page. For destination-unaware bursts, the least number of bursts per website can be 1, which would encompass all packets required to load a web page. For destination aware bursts, the number of bursts is no less than the number of IP addresses required to load web page, since a single web page might need resources from different servers.

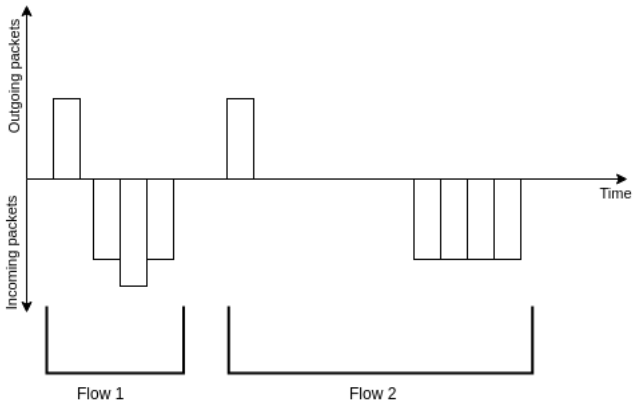


Figure 10. Packet flows example. Each flow is a cycle of packets sent and received.

Flows are defined by the same tuple as in previous paragraph (2.2, and bursts by tuple 2.4. Burst has additional variable - threshold $t$ which defines maximum delay between packets. Delay between packets is greater than $t$ marks a beginning of a new burst.

This paper will use two type of bursts - destination-aware and destination-unaware, both types without flows. Destination-aware bursts will be extracted per connection.

$$B = (t, s, r) \qquad C = (d, B_0, B_1, ...B_i) \tag{2.4}$$

Amount of available information that can be extracted about network traffic flow depends both on the application layer protocol, and lower level transport. For example, source and target IP

information of individual connections becomes unavailable when traffic is encapsulated in a VPN protocol, such as OpenVPN.

## 2.6 Fingerprinting Process

Fingerprinting usually refers to a process of mapping arbitrary length input into fixed size output. In our case we are trying to map traffic generated by a web page into a smaller size sequence of data points that can be later classified. Data points will be information extracted with a method selected from previous section. The information will be number of bytes sent and received.

Since it is not expected that all traffic captures will be identical, due to network conditions, and due to dynamic nature of websites, it is very likely that each capture will produce slightly different fingerprint. To deal with this issue, we can feed fingerprints with corresponding labels (website names) into a classification algorithm, and use it to predict to which website a specific traffic pattern belongs to.

There are two issues we must deal with. First is that number of websites we want to fingerprint is very small compared with number of all websites. This means that is use visits website which we did not fingerprint, our classification algorithm will still attempt to assign unknown fingerprint to one of the trained classes. To deal with this problem, a classifier can be modified so that it reports results only if prediction has some confidence score higher that selected threshold.

Another problem is that classification algorithms we want to use (eg. random forest) might require fixed size input. Since number of data points generated by different websites will differ, those data points cannot be directly used for classification. To solve this issue we can train many separate classifiers that for a range of different input sizes. Alternatively we can encode extracted features into fixed-length inputs for classification, for example, we can use statistical features of data points, and by calculating $N$ statistical features we will get $N$ features to feed into classification algorithm. In some scenarios it is valid to find label with largest number of features, and zero-pad shorter ones. In this case it will not be used. Multiple captures of the same website can produce fingerprints of different length, so it is assumed that statistical features would be less sensitive to such changes. Those features then can be used to train a classifier so that new fingerprints can be recognized. Some examples of statistical features are mean, standard deviation, maximum value. So the process can be divided into following steps, for each website to be fingerprinted:

1. Collect sample packet captures of a website.
2. Extract website traffic fingerprint from the packet capture (using flow information, bursts etc).
3. Normalize the data for classification algorithm.
4. Train the classifier.
5. Use trained classifier to predict which website's traffic matches traffic fingerprint.

This paper will target website fingerprinting by classification of fingerprints with random forest classifier, with single classifier being able to perform multi-class classification. This partially is a subset of research as done in paper by Taylor et al.[20] but focusing mainly on HTTP/3, with other methods and protocols provided as comparisons. Also, diverging from the paper, bursts and flows will be used as a separate entities, and burst will not be split into flows. Other papers

were considered as basis for this paper, such as k-fingerprinting[8]. Appscanner is simpler while still providing reasonable fingerprinting accuracy. Moreover, techniques from Appscanner can be applied to k-fingerprinting as algorithm in k-fingerprinting also uses random forest classifier. In case of k-fingepritning, random forest classifier does not yield final result but its properties are used for further processing.

## 2.7 Evaluation Metrics

Classification algorithms have multiple metrics which are derived from four possible outcomes of classification. The following outcomes are possible is we have we want to classify input as true or false:

- True positive (TP). True positive is classification result, where classifier identified input is true, and it is actually true.
- True negative (TN) - classifier identified input as false and it is actually false.
- False positive (FP) - classifier identified input as true while actually it is false.
- False negative (FN) - classifier identified input as not false but it is actually true.

This basic logic can be extended to multiple classes, not just to binary classification.

The following metrics will be found in experimental part:

- Accuracy.
- Precision.
- Recall.
- F1 score.

They are calculated from four possible outcomes of classification: Number of true positives $TP$, true negatives $TN$, false positives $FP$ and false negatives $FN$.

Formula for accuracy is

$$A = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.5}$$

Accuracy represents represents ratio of correctly predicted results out of all results. Accuracy will not be used as main metric as results may be unreliable when having imbalanced classes.

Precision gives a ratio of true positives out of all positives:

$$P = \frac{TP}{TP + FP}. \tag{2.6}$$

Recall similarly gives a ratio of predicted true positives out of all actual positives, and is defined as shown below

$$R = \frac{TP}{TP + FN} \tag{2.7}$$

Metric to be most used will be F1 score, which is derived from precision and recall:

$$F1 = 2 * \frac{P * R}{P + R} \tag{2.8}$$

These basic principles apply to multi-class classification. Experimental part of this paper will use macro averaging, as it is suitable for balanced classes. Classes will be balanced singe each website will be fingerprinted the same number of times, providing equal number of samples for each website.

# 3 Evaluation

## 3.1 Experimental Setup

Experimental setup starts with collection of website domains that support HTTP/3. According to the Web Technology Surveys[3] only about 3 percent of all websites use HTTP/3. Low percentage complicates collecting those domains for further analysis, as taking top 500 or similar lists would not result enough domains with HTTP/3 support. To find sizeable sample of domains of at least 100 that meet requirements, list of top 1 million domains was used[4] with small fraction of additional domains from Web Technology Surveys. Each domain was scanned by a custom script, which checked if domain supports h3-28 draft of HTTP/3. Although initial list of domains is large, but impact, while scanning, on each domain is minimal, as only main page of each domain was requested. A list of different domain name servers used to prevent inadequate load on default domain name servers. This resulted in a data set of nearly 600 domain names, which was further reduced by removing some domains, such as Google search domains names of different countries. Additional filtering was done to remove domain names that point to the same website, eg. `ampproject.net` and `ampproject.net` redirect to the same website. It was difficult to collect better data set, primarily due to low usage of HTTP/3 and the fact that most domain names supporting HTTP/3 are owned by Google. Google search domains `google.TLD` (TLD stands for top level domain), are identical from the content perspective, because Google localizes search page content by approximate user's location (eg. by IP address, locale settings).

To extract fingerprints from collected domains, development of data processing pipeline followed. It allows extracting information from packet captures generated by particular websites (fig. 11). Pipeline here is defined as a sequence of data inputs processed by applications that produce output for the next application.

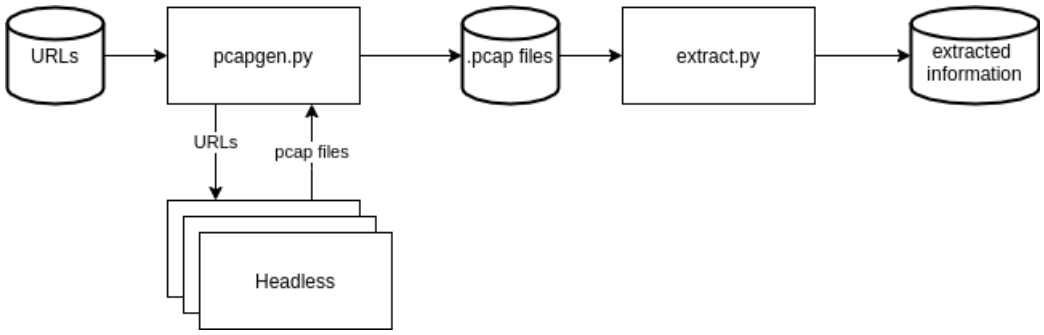


Figure 11. Information extraction pipeline.

Starting from left to right, the steps are as follows:

1. URLs database - list of collected domain names that support HTTP/3.
2. `pcapgen.py` - a script which takes a list of URLs and calls Headless service (step 3) to generate packet capture (`.pcap`) files. Also provides additional parameters to Headless, such as whether to keep connection open for HTTP/1.1, or whether to use another HTTP version.

[3] https://w3techs.com/technologies/details/ce-http3
[4] https://majestic.com/reports/majestic-million

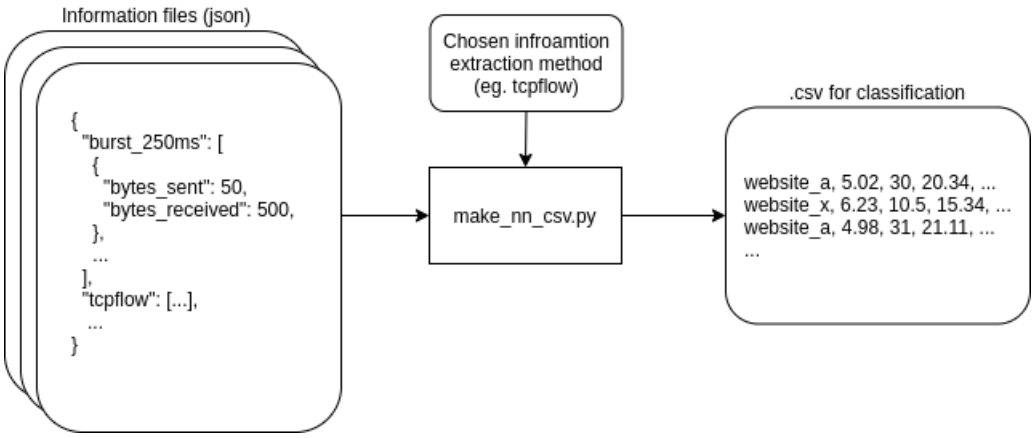


Figure 12. Generating data set for classification.

3. Headless - a service which runs in a docker container and captures network activity for given website. This is done by `pyppeteer` browser automation framework, with additional code for controlling other required parameters. Multiple instances are run at once for paralelizing packet capture process.

4. `extract.py` - extracts information from packet capture file. Multiple modes of operation are used, which allow extracting information from TCP connections, UDP connections for HTTP/3, burst, flows etc. This step finishes information extraction pipeline, as other tools will be used with visualization, and analysis. Information is outputted as a `.json` file.

URLs database contains 109 URLs, all of which have HTTP/3 support (at least draft h3-28). Packet capture using headless browser service was organized into runs, in each run each URL was accessed via headless browser four times (twice for HTTP/1.1, once for HTTP/2 and HTTP/3). There were 40 runs, which means that 40 packet captures were made for each URL per HTTP protocol. Extracted information files are structured as JSON, generated individually for each packet capture file. An excerpt of ‘https://abx.xyz‘ for HTTP/2 protocol is below:


```
{
 "tcpconn": [
  {
   "start_timestamp": 1607279944,
   "client_ip": "172.22.0.2",
   "server_ip": "216.58.215.78",
   "bytes_sent": 3367,
   "bytes_received": 202031
  }
 ],
 "tcpflow": [
  {
   "start_timestamp": 1607279944,
   "client_ip": "172.22.0.2",
   "server_ip": "216.58.215.78",
   "bytes_sent": 1476,
   "bytes_received": 8550
```

| | TCP connection | TCP Flow | Packet Burst (100ms) |
|---|---|---|---|
| HTTP/1.1 w/o keepalive | 17 | 83 | 15 |
| HTTP/1.1 | 7 | 49 | 13 |
| HTTP/2 | 7 | 42 | 13 |
| HTTP/3 | 7 | 32 | 14 |

Table 1. Average number of data points per protocol extracted by with different methods.

```
},
```

The idea behind information files is that they can be used for generating files to be used as input into chosen classification algorithms. Since packet extraction over a large initial packet capture files data set takes time, this is used as speedup for later development. So it was decided that the best idea is to process each packet capture only once. Note that only `bytes_sent` and `bytes_received` values will be used as classification inputs. Timestamp and IP values were included only for debugging.

Further processing involves formatting information so that it can be easily ingested by a classification or other chosen algorithms (fig. 12). Following experiments will use random forest classification algorithm.

Majority of code has been written in Python programming language, targeting version 3.8. Several bash scripts were used, mainly for gluing together, or for batch executing python scripts. Information extraction from packet capture files was done with `scapy` package with custom code. `pandas` package was used for preparing data for input into classification algorithm. Classification algorithm was used from `sciki-learn` package. Packet capture was performed with `tcpdump` and `pyppeteer` browser automation framework was used for automating website scraping. All graphs were rendered with `matplotlib` Other python packages were used, which can be found in corresponding `requirements.txt` files but mainly as a tooling to write as little boilerplate code as possible.

## 3.2 Exploratory Data Analysis

### 3.2.1 Data Points Overview

To understand data better we can compare an average number of data points extracted per website fingerprint. Different information extraction method produce different number of data points between protocols and different information extraction methods.

There are other features such as source and destination addresses but they were used only for debugging, and are irrelevant in further analysis.

Table 1 shows that using HTTP/1.1 without connection reuse (by injecting `Connection: close` header) produces most data points. It is explained by the fact that each resource is downloaded over an individual connection. It is included here only as a reference because it does not reflect real world traffic, and will not be included in further comparisons.

Number of TCP connections does not change between protocols. This can be explained that although protocols are different but the content is served from the same servers. The fact that HTTP/3 has TCP connections even though it is a UDP-based protocol might be confusing but it is explained by the protocol negotiation between client (browser) and server. Typical browser will first decide whether to use HTTP/1.1 or HTTP/2 protocol. This happens during TLS handshake

where client and server via ALPN TLS extension might indicate that they support HTTP/2. Once protocol is negotiated, and headers of the first request are received, the browser might switch to HTTP/3 by checking for `Alt-Svc` header. If header is present, it might indicate that some draft of HTTP/3 is supported, for example `h3-29`. So at least during start, the TCP connection is not avoidable, unless the browser decides to connect directly via HTTP/3 protocol.

TCP flow data shows that the least most data points are extracted with HTTP/1.1. HTTP/2 has less data points which is indication of HTTP/2 server push feature where server sends resources before client requests them. HTTP/3 has least data points but still a substantial amount. There are two factors here. First, not all website request have to be served by the same protocol, because not all servers might have HTTP/3 support, and not all of them need to have the same HTTP/3 draft version, since the HTTP/3 protocol is not finalized yet. Second, a browser can receive the first response via HTTP/2 since it already opened a TCP connection, and proceed loading other resource via HTTP/3.

Analyzing packet bursts results in least variation between protocols. Number of data points on average almost does not change, so it can be considered the most stable method. This suggests possibility of website fingerprints being protocol-independent when extracted with this method.

### 3.2.2 Packet Burst Threshold

Table 1 contains one important fact about packet bursts - packet burst threshold. Packet burst threshold defines a minimal time interval between packets that marks a beginning of a new burst. This means that by changing burst threshold we can control number of data points we want to extract.

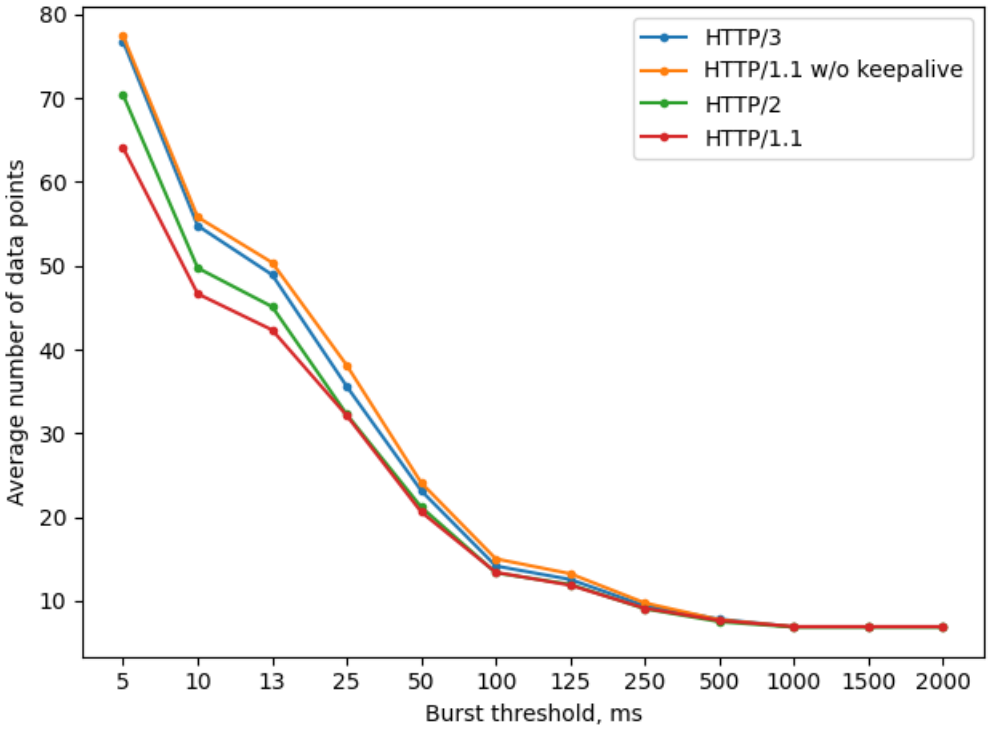


Figure 13. Number of bursts vs. burst threshold.

Figure 13 confirms that there is little variation between protocols with any burst threshold. Also, number of data points stabilizes when burst threshold is more than or equal to 1000 millisec-

onds (1 second). This is due to the fact that bursts are calculated per destination address. This means that for each destination address there will be at least one burst data point, and this is close to the average number of TCP connections.

There is one caveat with bursts. They are dependent on the connection speed at least to some degree. For example, given very slow connection and small burst threshold enough time can pass between request and a response, and they would appear in different bursts. With fast connection request and response may fit into one burst. Further tests will use burst threshold of 100 milliseconds.

As discussed in section 2, bursts can also be destination-unaware. This way no individual connections are considered, only packet timing, with minimal filtering, such as by transport layer protocol port. This results in far fewer bursts, as seen in fig. 14. What also can be seen is that all protocols result in almost the same number of bursts.

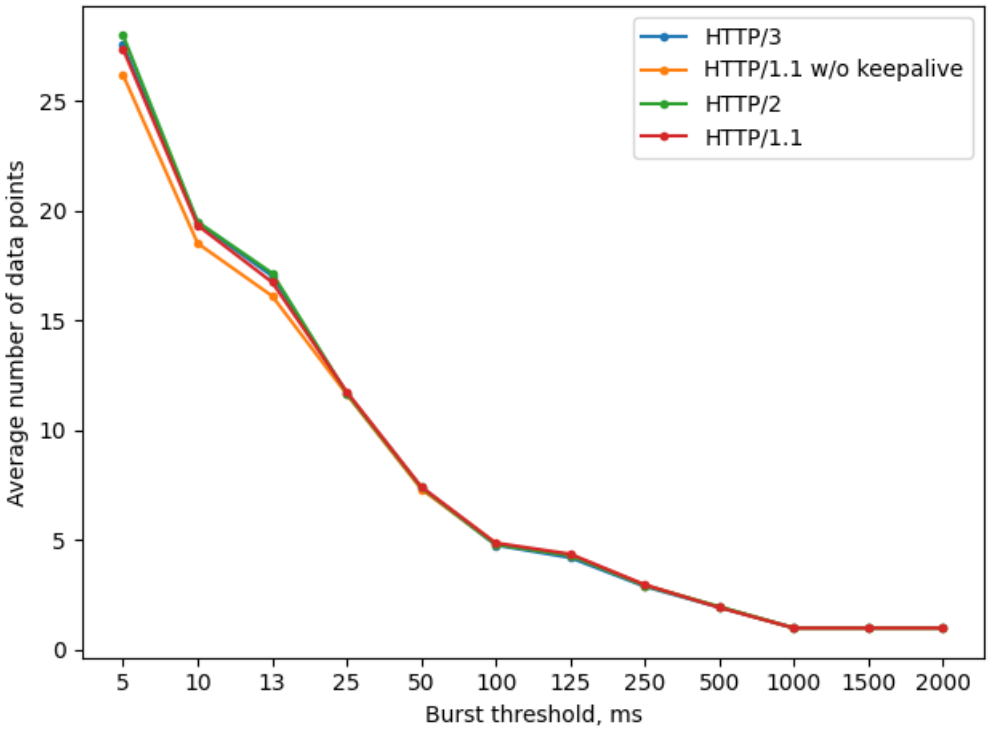


Figure 14. Number of bursts vs. burst threshold, destination-unaware.

The question is how to select a burst threshold value. Too short burst might create too much variation in number of bursts due to random variables, such as network conditions or server load, or even browser responsiveness. Too long bursts will not provide enough data points. To solve this, we can create multiple captures of a selected website traffic, then analyze each capture with a set of burst thresholds varying from a few milliseconds to a few seconds. Then we can find smallest burst threshold at which number of bursts does not change between captures. For this https://novi.com/ will be used, which is one of the websites in the list of websites that support HTTP/3. Sample size - 10 captures.

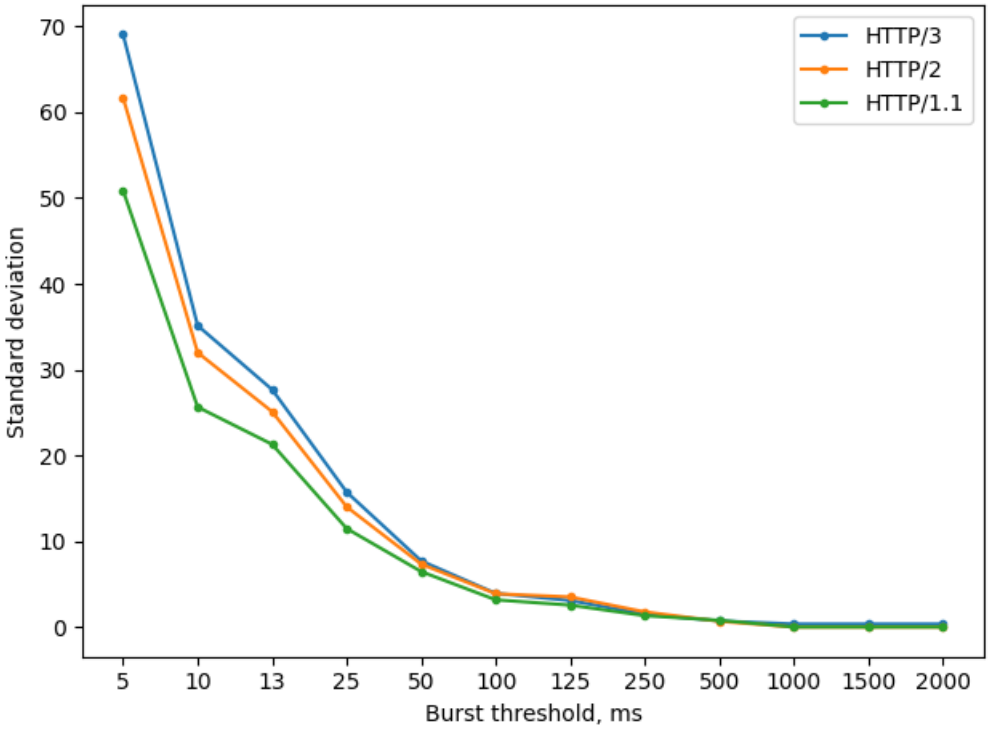


Figure 15. Standard deviation of number of bursts

We can see that standard deviation drops rapidly with increase of burst threshold interval. Also HTTP/3 has greatest variation on small burst thresholds but all protocols converge. Since website detection algorithm should tolerate some variation in the fingerprint, we will use burst threshold of 100 milliseconds .

## 3.3 Fingerprint Classification

### 3.3.1 Initial Results

We will attempt train random forest classifier in order to recognize fingerprints of websites. Data set will consist of 100 websites, each having 40 captures. Data set prepared for classifier will be a `.csv` file. First column will be label - website name, following columns will be data points. Example of data set:

```
freebasics.com, 1149976, 10615, 1.966902614036985, [...]
area120.com, 3251855, 7161, 2.891106873313672, [...]
```

Data set is generated from extracted information of packet capture. When generating data set for classification, a protocol and an information extraction method is selected, so the classifier is trained to recognize web pages that were accessed via chosen protocol. Also it follows that we must use the same information extraction method for training and actual classification attempts.

For initial test, the following information extraction methods were used for parsing packet captures:

- `tcpconn` - collecting how many bytes were sent and received per each TCP connection.
- `tcpdest` - collecting how many bytes were sent and received per each destination IP, monitoring only TCP traffic.

| Protocol | Method | Accuracy | Precision | Recall | F1 score |
|---|---|---|---|---|---|
| HTTP/1.1 w/o keepalive | tcpconn | 0.84 | 0.84 | 0.84 | 0.83 |
| | tcpdest | 0.94 | 0.94 | 0.94 | 0.94 |
| | tcpflow | 0.86 | 0.86 | 0.86 | 0.85 |
| | burst_250ms | 0.84 | 0.85 | 0.84 | 0.83 |
| | noconnburst_250ms | 0.87 | 0.87 | 0.86 | 0.84 |
| HTTP/1.1 | tcpconn | 0.89 | 0.89 | 0.89 | 0.88 |
| | tcpdest | 0.95 | 0.94 | 0.94 | 0.94 |
| | tcpflow | 0.84 | 0.84 | 0.85 | 0.84 |
| | burst_250ms | 0.80 | 0.80 | 0.80 | 0.79 |
| | noconnburst_250ms | 0.79 | 0.82 | 0.79 | 0.78 |
| HTTP/2 | tcpconn | 0.96 | 0.96 | 0.95 | 0.96 |
| | tcpdest | 0.95 | 0.95 | 0.95 | 0.95 |
| | tcpflow | 0.90 | 0.91 | 0.90 | 0.90 |
| | burst_250ms | 0.83 | 0.82 | 0.82 | 0.81 |
| | noconnburst_250ms | 0.84 | 0.82 | 0.80 | 0.79 |
| HTTP/3 | tcpconn | 0.93 | 0.92 | 0.92 | 0.92 |
| | tcpdest | 0.92 | 0.92 | 0.91 | 0.91 |
| | tcpflow | 0.88 | 0.89 | 0.88 | 0.88 |
| | anyflow | 0.86 | 0.85 | 0.86 | 0.85 |
| | udpflow | 0.69 | 0.69 | 0.69 | 0.67 |
| | udpdest | 0.92 | 0.93 | 0.93 | 0.93 |
| | burst_250ms | 0.76 | 0.76 | 0.77 | 0.76 |
| | noconnburst_250ms | 0.84 | 0.81 | 0.82 | 0.79 |

Table 2. Classification results using Random Forest classifier.

- `tcpflow` - collecting how many bytes were sent and received per each TCP flow. Only packets carrying application data are considered.
- `burst_N` - collecting how many bytes were sent and received per burst, taking into consideration both TCP and UDP. $N$ stands for burst threshold.
- `noconnburst_N` - same as `burst_N`, but bursts are not calculated per each destination IP.
- `udpflow` - collecting how many bytes were sent and received per each TCP flow.
- `udpdest` - collecting how many bytes were sent and received per each IP, monitoring only UDP traffic.
- `anyflow` - combining `tcpflow` and`udpflow` into one.

`udpflow` and `udpdest` are essentially targeted at HTTP/3, and are expected to provide identical results to their TCP versions.

Let's analyze the first classification attempt results shown in table 2. We will use F1 score as the main measure.

| Website | Precision | Recall | F1 score |
|---|---|---|---|
| adwords.com | 0.67 | 0.80 | 0.73 |
| ai.facebook.com | 0.73 | 0.89 | 0.80 |
| ampproject.org | 1.00 | 1.00 | 1.00 |

Table 3. Per-website result sample (HTTP/2 tcpflow).

Starting with HTTP/1.1 without keepalive, best results are achieved when we classify by information on how many data were transmitted per IP address. TCP flow and burst information perform about the same.

Surprisingly there is not much difference whether connection persistence is used or not, mainly that observing packet burst gives slightly lower score.

On HTTP/2 protocol best results are achieved with `tcpconn` with `tcpdest`. Using TCP flows is worse only marginally, by 0.05 when compares to `tcpdest`

Results with HTTP/3 protocol are the most surprising, and there are two important points to consider. First, we have used information extracted about TCP connections, and turns out those score in the range of $[0.88; 0.92]$. After looking into packet captures more closely, it was found that when HTTP/3 is enabled in browser, required resources loaded in a mix of HTTP/3 and HTTP/2.

For example, https://ampproject.org mixes HTTP/2 and HTTP/3 even when resources are served from the same domain. First hypothesis was that the website uses some for of load balancing, and some only some servers have HTTP/3 support enabled. Checking requests via browser showed that Chrome browser decided to use HTTP/3 only for some requests. This has implication that website fingerprints via HTTP/3 protocol only might be less stable than thought, if internal browser acts randomly. Or it might be HTTP/3 implementation detail, since it is not finalized. Another category of websites is, where main page document loaded via HTTP/2, and all other resources via HTTP/3, see https://daulaykausar.com/ as an example. Last observation is that Chrome browser can cache information about supported protocols. Opening https://http3check.net/ for the first time, main document is loaded via HTTP/2. Repeating this results in main document being loaded via HTTP/3. These facts can explain overall lower classification scores when HTTP/3 protocol is used. Lastly, it seems that no website reports HTTP/3 support via ALPN.

Another type observation to consider is that using UDP flow information results in worst score out of all results. There is one difference, however, between TCP flow and UDP flow information. In TCP flows we only consider packets that carry application data. Meanwhile with HTTP/3 UDP packets we cannot do that because all packets are encrypted.

As of bursts, it should be noted that when extracting information about packet bursts, the difference between extracting burst per IP address versus all network traffic, is insignificant.

One of the problems that were discovered with testing HTTP/3 was that not all websites which support HTTP/3 advertise that fact to the browser. The headless browser version used does not have an option to attempt use HTTP/3. Instead browser can only be instructed to prefer HTTP/3 if website advertises the protocol via `Alt-Svc` header.

Lastly, it must be noted that table 2 shows results that are averaged-out. How well an individual website can be classified varies, as shown in table 3

Overall results are close to those in Appscanner, where accuracy of "Single Large Random Forest" (random forest multiclass classifier whose inputs are statistical features of packets) is 86.9%. Differences arise due to different interpretation of bursts and flows, and due to different data sets.

### 3.3.2 Adjusting UDP flows

Using UDP packet flow information to classify website traffic showed the worst results. This is unexpected at first, since HTTP/3 and HTTP/2 are similar, except the underlying transport. However with `tcpconn` we consider packets only that carry application data, so packet types such as `ACK` are ignored. This is not possible with udpflow because UDP itself does not have acknowledgement mechanisms. The QUIC protocol draft states that "QUIC authenticates all packets and encrypts as much as is practical"[12]. Exact size of ACK frames can vary, due to the way acknowledgements work in QUIC - a single QUIC frame can acknowledge one or more packet ranges. Still, we can assume that majority of those packets will cary just one range of acknowledged packets and will be small on average, and so we will ignore them when extracting packet flow information.

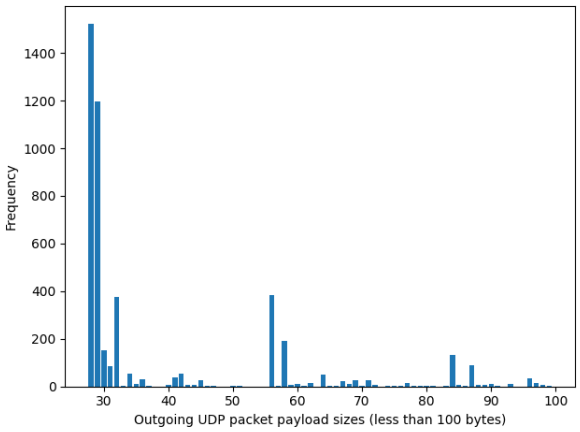


Figure 16. HTTP/3 outgoing UDP packet sizes.

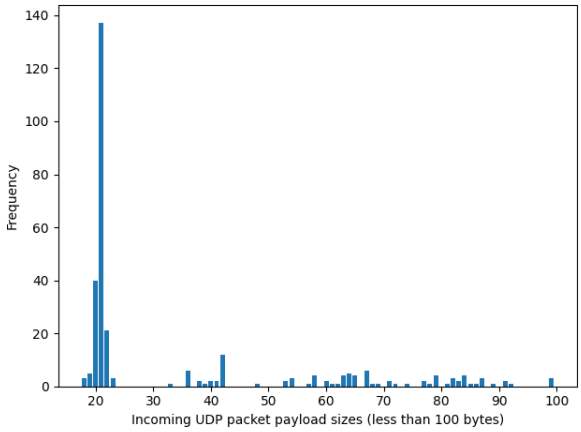


Figure 17. HTTP/3 incoming UDP packet sizes.

Graph 16 shows distribution of outgoing (sent from client) packet payload sizes that are less than 100 bytes. These were calculated from 109 packet captures, 1 per each website in our domains list. Outgoing packet sizes frequencies are similar, as shown in fig. 17. We can see that packet payload sizes are distributed unevenly, with a lot of occurrences of packet payload sizes between 28 and 33 bytes, and less but still significant occurrences of payload sizes of 56 and 58 bytes. For start we will extend `udpflow` information extraction method so that it can ignore small packet payload sizes. The minimal UDP packet payload size that breaks a flow will be called a flow threshold. They still will be included in overall number of bytes sent, but will not break a flow. Overall flow extraction algorithm changes minimally, as shown in fig. 18.

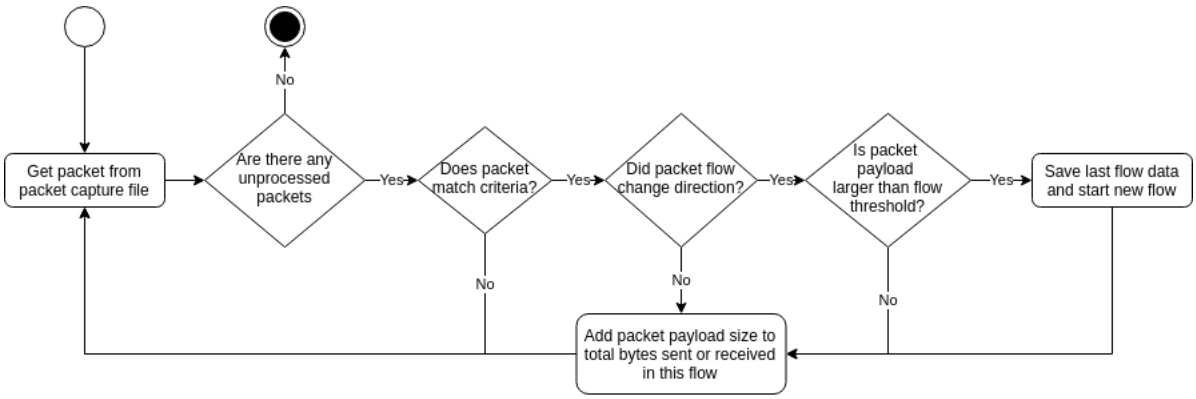


Figure 18. Flow extraction with flow threshold. See fig. 8 for comparison.

We will compare three cases - flow threshold of 0, 34, and 59. Flow threshold of 0 matches HTTP/3 `udpflow` row in table 2 but is included here for comparison.

| Flow threshold | Accuracy | Precision | Recall | F1 score |
|---|---|---|---|---|
| 0 | 0.69 | 0.69 | 0.69 | 0.67 |
| 34 | 0.79 | 0.80 | 0.80 | 0.79 |
| 59 | 0.84 | 0.84 | 0.84 | 0.83 |

Table 4. Results when using different flow thresholds for HTTP/3.

In table 4 we can see that with increase of flow threshold our classification results become more accurate overall. This means that we were able to filter out QUIC packet transmission acknowledgement frames, and fingerprints generated from packet captures became more stable. We can try more flow thresholds and see how do they perform, and find an optimal flow threshold. Figure 20 shows that selected flow threshold values were correct guesses, as F1 score does not increase with larger flow threshold values. This means that noise caused by QUIC protocol was successfully filtered, and that considering QUIC acknowledgement packets is an important step during packet capture information extraction.

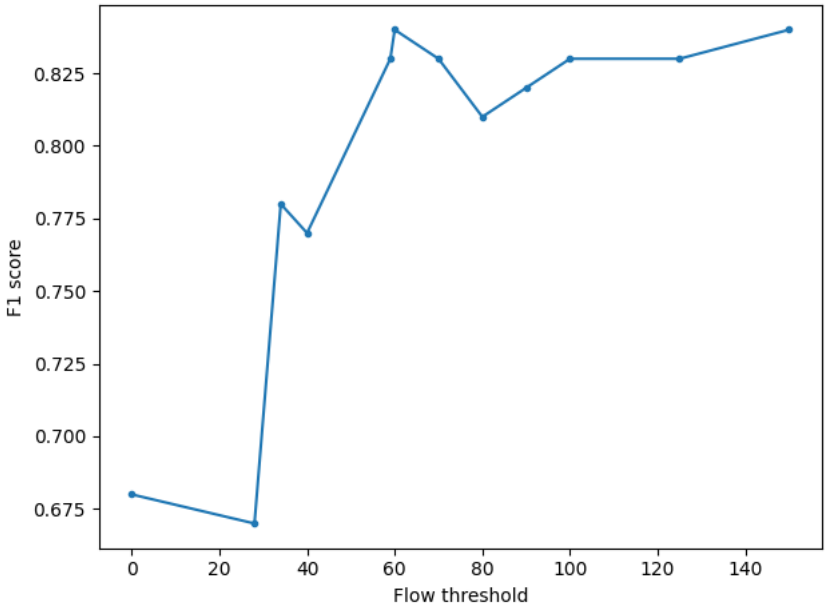


Figure 19. F1 score with different flow threshold values.

### 3.3.3 Adjusting Burst Threshold

Having introduced flow threshold and noticing its affect on classification results, we can check another variable that we can change - burst threshold. Although a best value value was estimated in previous sections, we can try a range of its values.

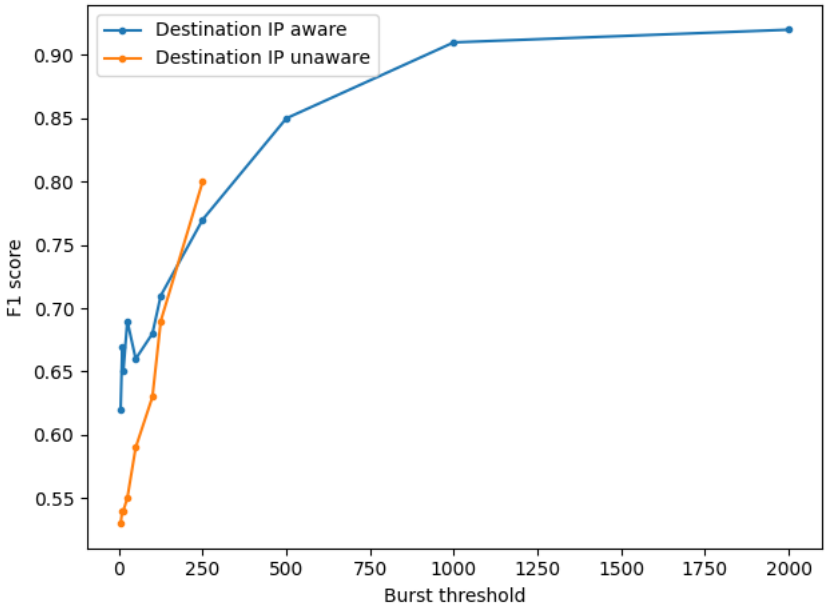


Figure 20. F1 score with different burst threshold values.

Graph 20 shows that the bigger burst threshold, the better are classification results. However, in case of connection-aware bursts, with bigger burst threshold all packets fit into a single burst per destination IP, and this essentially equals calculating how many bytes were sent and received per destination IP. This is essentially matches `tcpdest` method.

Note that there is no data for destination IP unaware bursts (`noconnburst`). This is because there were no cases when there is period of inactivity in packet capture where there were no packets transmitted for longer than 250 milliseconds. This resulted in a single burst. Single burst, being a single data point, was not enough to calculate statistical information. In result, classification could not be performed.

### 3.3.4 Adjusting Number of Features

Input to classification algorithm is derived from extracted information statistical features. Twenty-five features were used, a subset of features used by Taylor et al.[20]. First ten features are listed as follows:

1. Maximum value out of bytes sent or received.
2. Maximum value out of bytes sent.
3. Skew of bytes sent and received (values of bytes sent and received are combined into one list).
4. Variance of bytes sent.
5. Standard deviation of bytes sent.
6. Kurtosis of bytes sent.
7. Skew of bytes sent.
8. Median absolute deviation of bytes sent.

9. 90th percentile of bytes sent.

10. Mean of bytes sent and received.

The number of values chosen was arbitrary. We can check if reducing number of statistical features impacts classification results. The goal would be to minimize number of these features because it would result in less computations required. In graph 21 we can see how number of statistical features given given for classifier affect results. We can see that having just three statistical features results in F1 score close to 0.8. Also noticeable that with HTTP/3 more statistical features are needed to achieve F1 score comparable to other protocols. The conclusions are valid in this case where we have a small number of websites. With larger data sets this might cause different results.

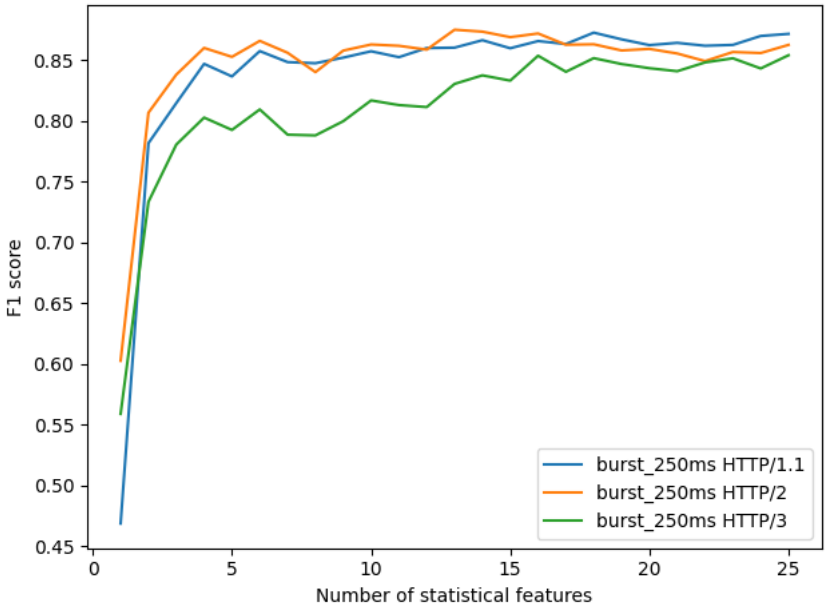


Figure 21. F1 score with different number of features provided to classifier.

# Conclusions and Recommendations

Applying website fingerprinting methods to HTTP/3 turned out to be the easier part out of all preparations and research. Out of all main point to conclude, it should be stated that HTTP/3 protocol is not finalized yet. This leads to the fact that even though it is not expected to change significantly, some details may change and this could affect fingerprinting, for example, due to implementation details, such as padding. Second, it is difficult to independently collect a large number of domains that support HTTP/3. Even when collected, most of them being owned to Google, and are duplicates that target mainly Google search, Youtube, and other Google services. Finalization of the protocol and time solve this issue, as browsers already have support for the protocol even if it is disabled by default, and some website owners experiment by enabling the protocol.

During development of experimental part it was discovered that some websites that support HTTP/3 do not advertise that information yet, which meant it was not possible to access them via browser with HTTP/3 protocol. Also, many websites are loaded via multiple protocols due to the fact that resources are loaded from multiple web servers. Not of all are HTTP/3-enabled. This means that multiple protocols must be considered when analyzing packet capture with intention of fingerprinting websites visisted.

Finally, when extracting information from a packet capture, connection-oriented methods are not applicable due to usage of UDP. Other methods such as extracting packet flow information must be adjusted to ignore QUIC protocol packets that carry information such as packet acknowledgements. In results, this was accomplished by packet size frequency analysis and allowed to improve classification results.

Overall, usage of HTTP/3 does no significantly decrease classification results. This paper being an early attempt of researching HTTP/3 impact, could be revisited when HTTP/3 gets finalized and more widespread, and more data will be available.

Lastly, even though this research focused on a limited set of features and algorithms, knowledge about HTTP/3 could be applied research of other papers.

# Future work

There are two valuable directions for future work - trying different algorithms for classification and looking into other authors research into details, or providing a practical implementation that could be used as a framework. For example by a network administrators wanting to identify unwanted activity.

Implementing this and similar research as a tool should result in a quite a modular design. The usual workflow is that packet captures are collected for analysis and algorithm training, followed by information extraction from collected packet captures, then training, evaluation. Finally an actual packet capture of some user's traffic can be analyzed, and predictions made on what websites were visited.

Each mentioned step can be thought as a separate module. For example packet capture collection is completely independent of next steps. As it was done in experimental part of this paper, docker images running a headless browser software are a good choice for this. Collection of packet captures can be extended to run on multiple machines for better throughput. Packet capture process should be improved so that it will not be stopped by website anti-scraping measures, which might distort fingerprint used for training versus actual website traffic.

In this paper different information extraction methods that extract information form packet capture were mostly implemented as a separate classes in Python code. Instead they can be treated as separate plug-in modules that can be used by framework. Some of these modules could be provided as default by the framework, while others could be supplied by user, and be enabled or disabled on further framework user needs. They also can be optimized for faster information extraction, and be rewritten in languages such as C, providing binding to they core written in Python. Experimental setup of this paper had information extraction step and another processing stage so that information could be fed into into classification algorithm, so these steps can be chained.

Same ideas apply to classification training and testing step. They could also be replaced, or chained.

Processing of real user traffic and fingerprinting traffic is more complicated. If we want to fingerprint websites visited by a user or multiple users on a network, we either capture packets and store them for later analysis, or analyze them in real time. Both mechanisms should function equivalently except that real time analysis would require fast processing to keep up with network speeds.

Classification in previous sections has been done with packet captures where only one website is accessed per packet capture. This makes information extraction, further processing, and classification simple but does not reflect real world scenario.

In classification training and testing information from whole packet capture file were used. When analyzing a packet capture with more data, sliding window approach can be applied.

This means that only a part of packet capture should be fed into classification algorithm, then part that has some offset from then start of the packet capture and so on. The idea behind this is that at each point in time it is likely that only one website will be accessed at a time. Although this is not always a case, for example if a PC user is streaming music and visiting some other website at the same time. A better case would be web browsing via smartphone, where user activity is more focused on a single activity at a time.

The sliding window can be defined a a time span. This time span limits how many packets are processed for feature extraction For example in a window all information from last five seconds

may be used. Exact time span should be defined experimentally by comparing classification results, or by finding average load time of websites that need to be identified.

Next, there must be a decision made at what resolution will the sliding window operate. Either sliding window steps per each packet, or per each unit of extracted information. The second case looks simpler since information extraction code would not need any changes but the problem is that some of our information extraction methods operate per whole packet capture, eg `tcpdest`, `udpdest`

As previously with extracted information from whole packet capture, statistical features can be derived form each sliding window of information and used as input to pre-trained classifier. Classifier would classify each window of extracted information. It is not expected that each window will be classified as there might be overlaps between requests to different websites, or periods of no activity However it is expected that classification will be robust enough to identify a website even if not a complete fingerprint of it is present in a single window.

# References


[1] mitmproxy is a free and open source interactive https proxy. https://mitmproxy.org/.

[2] Lenovo "superfish" controversy – what you need to know. 2015. https://nakedsecurity.sophos.com/2015/02/20/the-lenovo-superfish-controversy-what-you-need-to-know/.

[3] Chuvakin A. and C Peikari. *Security Warrior*. O'Reilly Media, Inc., 2004. page 229.

[4] Mohammad Al-Hisnawi and Mahmood Ahmadi. Deep packet inspection using quotient filter. *IEEE Communications Letters*, 20(11):2217–2220, 2016.

[5] Mohammad Al-Hisnawi and Mahmood Ahmadi. Deep packet inspection using cuckoo filter. In *2017 Annual Conference on New Trends in Information & Communications Technology Applications (NTICT)*, pages 197–202. IEEE, 2017.

[6] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, pages 44–51. IEEE, 2003.

[7] Ryan M Gerdes, Thomas E Daniels, Mani Mina, and Steve Russell. Device identification via analog signal fingerprinting: A matched filter approach. In *NDSS*, 2006.

[8] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 1187–1203, 2016.

[9] Andrew Hintz. Fingerprinting websites using traffic analysis. In *International workshop on privacy enhancing technologies*, pages 171–178. Springer, 2002.

[10] Paul E. Hoffman and Patrick McManus. DNS Queries over HTTPS (DoH). RFC 8484, October 2018.

[11] Martin Husák, Milan Čermák, Tomáš Jirsík, and Pavel Čeleda. Https traffic analysis and client identification using passive ssl/tls fingerprinting. *EURASIP Journal on Information Security*, 2016(1):6, 2016.

[12] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-33, Internet Engineering Task Force, December 2020. Work in Progress.

[13] Maciej Korczyński and Andrzej Duda. Markov chain fingerprinting to classify encrypted traffic. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 781–789. IEEE, 2014.

[14] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *ACM SIGCOMM Computer Communication Review*, 36(4):339–350, 2006.

[15] Weiran Lin, Sanjeev Reddy, and Nikita Borisov. Measuring the impact of http/2 and server push on web fingerprinting.

[16] Jianting Ning, Geong Sen Poh, Jia-Ch'ng Loh, Jason Chia, and Ee-Chien Chang. Privdpi: Privacy-preserving encrypted traffic inspection with reusable obfuscated rules. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1657–1670, 2019.

[17] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. TLS Encrypted Client Hello. Internet-Draft draft-ietf-tls-esni-09, Internet Engineering Task Force, December 2020. Work in Progress.

[18] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 213–226, 2015.

[19] Arun Subramaniyan and Reetuparna Das. Parallel automata processor. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 600–612. IEEE, 2017.

[20] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 439–454. IEEE, 2016.

[21] Charles V Wright, Lucas Ballard, Scott E Coull, Fabian Monrose, and Gerald M Masson. Uncovering spoken phrases in encrypted voice over ip conversations. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):1–30, 2010.

[22] Fang Yu, Zhifeng Chen, Yanlei Diao, Tamil V Lakshman, and Randy H Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, 2006.