Příjemci podpory:

Vysoké učení technické v Brně
Fakulta informačních technologií

Poskytovatel:

Ministerstvo vnitra ČR

***Integrovaná platforma pro zpracování digitálních dat z bezpečnostních incidentů* (TARZAN)**

Identifikační kód VI20172020062

Název předkládaného výsledku: DNS Over HTTPS Analysis and Detection

Typ výsledku dle UV č. 837/2017	Evidenční číslo (příjemce)	Rok vzniku
O ostatní	FIT-TR-2019-5	2019
ISBN-ISSN	**Webový odkaz na výsledek**	**Kde a kdy publikováno**
	https://www.fit.vut.cz/research/publication/12142/	Technická zpráva FIT VUT v Brně

Anotace k výsledku:

The DNS is an old protocol for the translation of domain names into IP addresses. The protocol was extended over the years. Still, it is primarily used in pure form without any encryption. There are attempts to make this protocol more secure by encapsulating it to different protocols. One of the last efforts is DNS over HTTPS. This work makes the analysis and detection of this protocol in regular HTTPS traffic.

Řešitelský tým: *Petr Matoušek (manažer a hlavní řešitel), Kamil Jeřábek (realizační tým)*

DNS Over HTTPS Analysis and Detection in Regular HTTPS

FIT-TR-2019-05

Kamil Jeřábek

Document for TARZAN project
Faculty of Information Technology, Brno University of Technology

Last edit: July 3, 2020

Detection of DNS Over HTTPS in Regular HTTPS

Kamil Jeřábek

Brno University of Technology, email: ijerabek@fit.vutbr.cz

Abstract. The DNS is an old protocol for the translation of domain names into IP addresses. The protocol was extended over the years. Still, it is primarily used in pure form without any encryption. There are attempts to make this protocol more secure by encapsulating it to different protocols. One of the last efforts is DNS over HTTPS. This work makes the analysis and detection of this protocol in regular HTTPS traffic.

1 Introduction

Most of the network traffic these days is encrypted. With the increasing tension to secure the systems and also the connections between those systems, we can expect that all traffic will be encrypted soon. Some of the protocols we are using these days are old protocols that do not count on the encryption in the protocol design. The other security protocols are used to encapsulate the unencrypted protocols and provide security and encryption to them.

Currently the most discussed protocol is DNS protocol[8, 9]. The DNS itself does not provide any encryption or security. Hence, it is open to multiple vulnerabilities and attacks. DNS Security Extension was introduced to add origin data authentication and integrity to the system[5]. This extension does not protect the privacy of the transferred data. Internet Service Providers (ISPs) and other authorities can still spy on the users, do filtering, and so on.

There are attempts to encrypt the DNS traffic by encapsulating the requests and responses into other protocols such as DNS over TLS[3], DNS over DTLS[11], the last and most discussed these days DNS over HTTPS[7].

The DNS over HTTPS (DOH) is the newest and most discussed attempt to encrypt the DNS traffic. Unlike the others, the DOH does not use its separated port so it can not be easily filtered, and the transferred data can be mixed with the regular HTTPS traffic. Spying on the user requests is pushed to resolver providers, that can also filter and deliver different services. The enterprise network administrators are losing control of even determining if the users are using the DOH or not.

This work aims to analyze the DNS over HTTPS traffic and the detection in regular HTTPS traffic. Because it is a new approach on how to protect the DNS traffic, it would be helpful also to compare this approach to the previous protocols and point to a fact what it brings. Machine learning algorithms are then used to do detection.

This document describes the analysis of the protocols and the attempt to do detection of the DOH in regular HTTPS traffic.

The document has the following structure: Section 2 describes the research method used in this work. Section 3 briefly introduces the structure a form of the protocols as they are defined in RFCs. Section 4 is focused on data generation of the dataset that is used for the analysis and further detection. Section 5 gives a short information about what system is used for processing such a huge amount of generated data. Section **??** gives the evaluation of the statistics extracted from the generated data, and it contains its comparison. The next 8 section describes how browsers are dealing with DOH implementation. The last section 9 before final conclusion focuses on DOH identification in regular flows. Section 10 contains final thoughts together with list of bibliography.

2 Research Method

The core of this work is to analyze DNS over HTTPS traffic and then try to detect it in the regular HTTPS traffic. It is not a trivial task, as the DNS can be transmitted as data over HTTPS, and it can also be mixed with other transferred data in the same session.

The research method covers data analysis and experiments. At first, the protocol data and the traffic data will be analyzed. Because machine learning will be involved, it is needed to generate a meaningful amount of labeled data for training and testing. The traffic will be generated as separated single DNS over HTTPS request responses. With the same process also regular DNS traffic and DNS over TLS request responses will be created for the comparison.

Once the generation is done, the pre-processing, cleaning, and filtering will be done. The datasets will be processed and analyzed using big data frameworks. Another dataset for the testing will be generated. To provide the most accurate results, the traffic for the testing will come from a regular browser.

An idea to do detection is to use statistics extracted from the first x packets and try to detect the occurrence of the signature in regular traffic. This approach is promising because the sessions with the resolvers can last very long. If the plan with the generated data will not produce satisfying accuracy, the data generated by the Firefox browser will be used for the training instead. The experiments will cover the training and testing of the created models. The accuracy will be determined.

3 Protocols

This section provides a brief description of the three protocols, namely pure DNS, DNS over TLS, and DNS over HTTPS. Those are the three chosen protocols for comparison and analysis, while the main point of interest remains DNS over HTTPS. The description is taken from the original RFCs and is mainly focused on the protocol structure and characteristics helpful for this work. The primary focus is on an exchange between clients and servers.

3.1 Domain Name System

The DNS protocol is defined in RFC 1035[9]. The DNS is a protocol for the translation of domain names into IP addresses. The communication uses messages whose top-level format is depicted in figure 1.

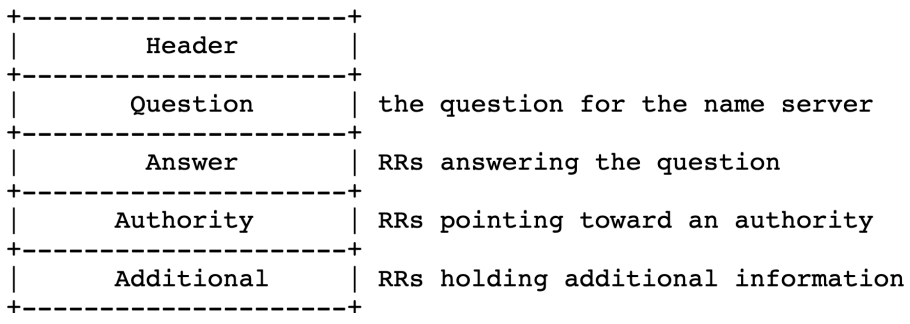


Fig. 1: DNS message format.[9]

The message is divided into multiple parts, where some of them are optional. The message always contains header determining if the message is a question or an answer and so on. The header follows the question part that carries the three fields, query type, query class, and query domain name. After this part, there is a possibly empty list of resource records with the Answer that answers the question, the Authority that points toward an authoritative name server, Additional section contains additional information related to the query, but not strictly answering the question itself[9].

The DNS can be transmitted using both UDP and TCP transport protocols. The reserved port is 53. The UDP payload messages are restricted to the size of 512 bytes. In case of longer DNS message, TC bit is set in the header, and the message is truncated. In the case of TCP protocol, the two-byte length field is added before the DNS message in TCP payload, giving the length of the message. The maximum TCP payload size can be used, and it is not limited to the only 512 bytes, as in the case of UDP.

3.2 DNS over TLS

The DNS over TLS protocol brings encryption and authentication to the DNS communication. It is defined in the RFC 7858 [3]. It uses standard port 853 as the default port for this protocol. The TLS protocol encapsulates DNS request/response as a payload as it is depicted in figure 2.

Two entities, client and server, are using the TCP transport protocol for data transfer. Hence, the communication starts with TCP handshake. Once the client creates a successful TCP connection, it proceeds with TLS Handshake defined in RFC 5246 [2].

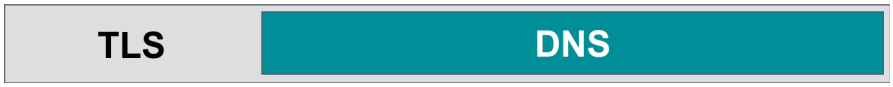


Fig. 2: DNS over TLS encapsulation.

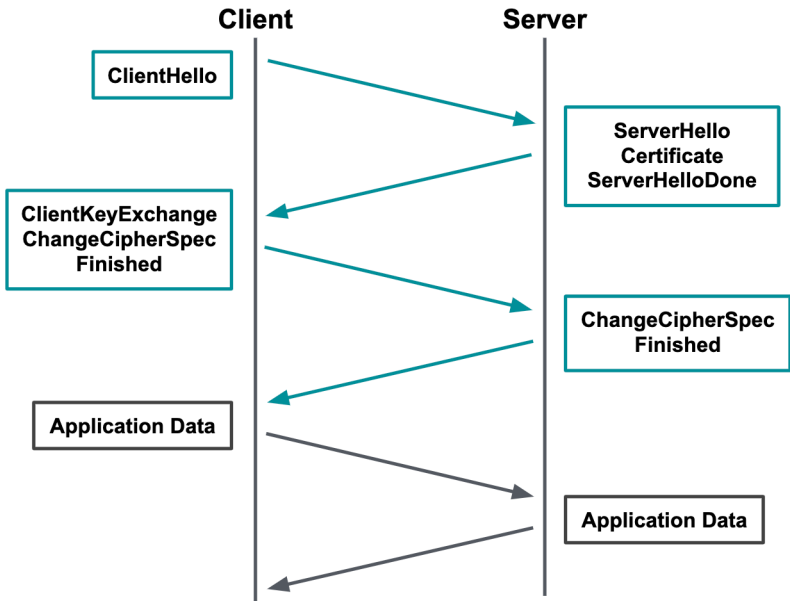


Fig. 3: TLS handshake.

The TLS handshake may contain less or more messages, depending on the agreement between client and server. The authentication continues if required, but it is not necessary. Once the TLS handshake and optional authentication is finished, the client may start sending requests to the server.

To minimize the latency, the clients do not have to wait for the responses, and they can pipeline multiple queries over one TLS session. The transmitted data are the same as in the case of DNS used with TCP transport protocol.[3]

3.3 DNS over HTTPS

DNS queries and responses are transferred using HTTPS on the known port 443 for HTTPS. This new protocol is defined in the RFC 8484 [7]. The following part describes the DNS data definition that comes from this RFC. The HTTP is protected with TLS. DNS query is then encoded using the HTTP POST or GET method. The DNS server always has to support both options. When using the POST method, no variables are accepted. On the other hand, the GET method accepts one variable that is `dns`. The figure 4 depicts the encapsulation of the DOH.

Fig. 4: DNS over HTTPS encapsulation.

The figure 5 depicts example POST DNS query. HTTP header contains request field `Content-Type` representing media type of value `application/dns-message`. The body of the message carry DNS query in the DNS wire format[9].

```
:method = POST
:scheme = https
:authority = dnsserver.example.net
:path = /dns-query
accept = application/dns-message
content-type = application/dns-message
content-length = 33

<33 bytes represented by the following hex encoding>
00 00 01 00 00 01 00 00  00 00 00 00 03 77 77 77
07 65 78 61 6d 70 6c 65  03 63 6f 6d 00 00 01 00
01
```

Fig. 5: DOH POST query example.[7]

The HTTP header may contain `Accept` field to clarify what type of response the client expects. The DNS query using HTTP GET is shown in figure 6. The HTTP queried domain is encoded in base64url format without padding as value of the `dns` variable.

```
:method = GET
:scheme = https
:authority = dnsserver.example.net
:path = /dns-query? (no space or Carriage Return (CR))
        dns=AAABAAABAAAAAAAAAWE-NjJjaGFyYWN0ZXJsYWJl (no space or CR)
        bC1tYWtlcy1iYXNlNjR1cmwtZGlzdGluY3QtZnJvbS1z (no space or CR)
        dGFuZGFyZC1iYXNlNjQHZXhhbXBsZQNjb20AAAEAAQ
accept = application/dns-message
```

Fig. 6: DOH GET query example.[7]

The DNS response should be of type `application/dns-message`, but the authors of RFC 8484 [**doh-rfc8484**] points to the fact that other formats could be introduced in the future. One DNS request and response should be mapped to one HTTPS exchange, but HTTPS multi-streaming functionality can be involved for the replies.

```
:status = 200
content-type = application/dns-message
content-length = 61
cache-control = max-age=3709

<61 bytes represented by the following hex encoding>
00 00 81 80 00 01 00 01  00 00 00 00 03 77 77 77
07 65 78 61 6d 70 6c 65  03 63 6f 6d 00 00 1c 00
01 c0 0c 00 1c 00 01 00  00 0e 7d 00 10 20 01 0d
b8 ab cd 00 12 00 01 00  02 00 03 00 04
```

Fig. 7: DOH response example.[7]

The data payload for this media type is a single message in the on-the-wire format, as it can be seen in the example picture 7. The maximum DNS message size is defined to be 65535 bytes. The fields from the DNS header are optional.

```
curl -H 'accept: application/dns-json' 'https://cloudflare-dns.com/dns-query?name=example.com&type=AAAA'
```

Fig. 8: DOH JSON request example.

The big DNS resolver providers such as Cloud Flare and Google implement the opportunity to exchange DNS information also using JSON format. This format is not included in the RFC. The `content-type` that the client is expecting is `application/dns-json`.

The request URI can be seen in figure 8. The main variables are `name` for the domain name in plain text and `type` for the DNS response.

The response can be seen in the figure 9. The response fields and information is similar for both Google and Cloud Flare providers, but the examples come from Cloud Flare developers documentation[1].

[1] https://developers.cloudflare.com/1.1.1.1/dns-over-https/json-format/

```
{
  "Status": 0,
  "TC": false,
  "RD": true,
  "RA": true,
  "AD": true,
  "CD": false,
  "Question": [
    {
      "name": "example.com.",
      "type": 28
    }
  ],
  "Answer": [
    {
      "name": "example.com.",
      "type": 28,
      "TTL": 1726,
      "data": "2606:2800:220:1:248:1893:25c8:1946"
    }
  ]
}
```


Fig. 9: DOH JSON response.

4 Data Generation

The main aim of data generation is to generate meaningful labeled traffic for the analysis and comparison of the mentioned protocols. And more importantly, it is focused on creating datasets for training and testing the machine learning models for the detection.

4.1 Single Request Response Data Generation

Meaningful source of domains is needed to generate requests. The domains come from the top one million domains from OpenDNS dataset from the date 1.5.2019 [].

At first, the filtering of those domains was done, and the domains were split into multiple categories. There are 3 categories based on the DNS response, such as normal single responses, responses that contain CNAMEs, responses that contain errors. The errors provide only responses that are not found or have a decoding problem. Those categories are created both for IPv4 and IPv6. The results of responses for every domain was taken using doh client tool[2] from curl for DNS over HTTPS. The domains were assigned to the categories based on the response message. The queries were done against the Cloud Flare cloud and DNS service provider.

The number of domains assigned to each category is shown in table 1. The total count of domains that were requested is 1463320, the both IPv4 and IPv6

[2] Simple DOH client written in C language supporting DOH post requests: https://github.com/curl/doh

Category	Domains
ipv4	515311
ipv4 cn	278910
ipv4 error	304
ipv6	92186
ipv6 cn	66775
ipv6 error	102160
both ipv4 and ipv6 error	203837

Table 1: Number of domains in each category.

error category is included once for both IPv4 and IPv6. It contains domains where there was an error in response to IPv4 and also IPv6.

The analysis focuses on a comparison of previously mentioned DNS protocols and the options currently provided by Cloud Flare and Google DNS Server and others. It is also focused on the traffic behavior of such protocols. Such options are DNS, DNS over TLS, and DNS over HTTPS. In the case of DNS over HTTPS, RFC 8484 [7] defines multiple options for requests, and DNS providers add one in advanced, a response in JSON format. Requests for all of those options were made using different publicly provided tools.

Data Generation Process The Docker containers were used for data generation. The docker provides network and application isolation. The isolation is an important aspect that enables the capture of clean network traffic from a container running only one application communicating over the network.

Various tools, kdig[3] for DNS and DNS over TLS, and doh tool from curl[4] and our written tool[5] for all kinds of DOH requests were used to generate single request responses of the protocols and their options. The tools are working correctly, and those full fill all the requirements defined in particular RFCs.

There was one container running sequential requests for each category. The order of the requests was always the same for every tool. The tcpdump[6] tool for capturing packets was used within each container. The Alpine curl[7] Docker container was used. The containers were run on the platform Supermicro SuperTwin2 6026TT-TF server equipped with eight Intel (R) Xeon E5520 @ 2.26 GHz. The cluster consists of 4 nodes, the nodes equipped with the 48 GB RAM, and 16 CPU cores. The distance from each node towards the DNS server was the same.

[3] Link to knot-dig man pages: https://knot.readthedocs.io/en/stable/man_kdig.html

[4] Simple DOH client written in C language supporting DOH post requests: https://github.com/curl/doh

[5] Simple DOH client supporting multiple types of requests: https://github.com/kjerabek/Simple-DNS-over-HTTPS-client

[6] https://www.tcpdump.org/

[7] Official image used: https://hub.docker.com/r/storytel/alpine-bash-curl

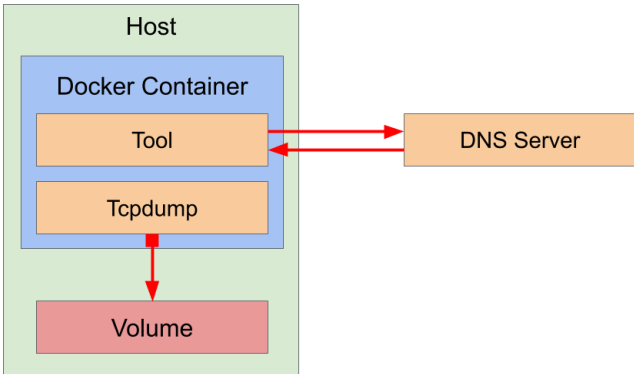


Fig. 10: Container structure for single request response data generation.

Single Request Dataset Summary The 1463320 requests were made for each provided application protocol option mentioned in this section. The captured data was processed and divided into separated requests. DNS over TLS and DNS over HTTPS protocols use TCP as an underlying transport protocol. Those requests were divided into TCP flows.

The flows were filtered, and only complete flows that contain all packets, including TCP handshake and finish, were included. The rest was discarded. The following table 2 provides the number of correctly processed flows. The extracted flows were later filtered for TCP RST flags.

Category	**TCP Flows**
Clear flows without RST flags	
DOH POST	1440743
DOH JSON	1463313
DOH GET	1463305
DOT	1462386
Flows with RST flags	
DOH POST	373
DOH JSON	0
DOH GET	0
DOT	2

Table 2: Resulting number of flows in dataset for each tool.

The pure DNS requests using the underlying UDP protocol were also made. The requests were processed and divided into separated requests responses by the source port and by the timing. The data was cleaned, and the result contains ..... separated requests.

4.2 Single Web Page Request Response Data Generation

The data source used for this generation was Majestic Million[6] free root domain list. The list contains top million root domains for websites by reference. This domain source is different than the previous one, so at least the main domains requested for resolution will be different.

The domains from chosen list were used to generate browser traffic by accessing the websites on those domains. Following paragraphs describes the whole generation process of this traffic.

Then the generated traffic was used for the purpose of Big Data analysis and also for training and testing Machine Learning models to be able to classify the DOH and regular HTTPS traffic.

Accessing most of the web pages the browser downloads various files such as `*.js`, `*.css`, `*.jpg`, `*.html` and so on. Most of the times it points to different servers. In terms of dynamic web pages the data can be fetched using REST API. So basically, to be able to load single web page the browser has to create multiple network flows mostly to different IPs. Because the source locations pointing to other servers are specified as URIs hence the domains has to be resolved by DNS. By using the DOH the browser has to create at least one network flow to DOH server and send multiple requests to be able to resolve those domains.

The DOH server used for resolution is known by the browser. Hence, the DOH traffic can be later distinguished from the regular HTTPS by DOH server IP address. Then, this approach is used for labeling the traffic into two categories, the regular HTTPS and DOH. The labeled and filtered data are used both for analysis and classification.

The principle is the same, but for different parts of the analysis there is used different number of web page queries.

Data Generation Process The Docker containers were also used for the data capturing in this case. The Docker container contains the Selenium script and Firefox browser installed together with `tcpdump` tool for capturing the traffic.

The Selenium script starts the browser in a clean state with enabled DOH usage and resolver set to the default Cloud Flare's IP address 1.1.1.1. Thanks to this, the DOH traffic can be filtered later. The Firefox browser is also set to produce log files with all HTTP/HTTPS headers from all request responses. Thanks to the logs, more statistics can be extracted from the collected data.

The container is started for every single domain from the domains list. This behavior creates pcap files and logs files that contain traffic from only one website traffic access. The container definition with the selenium script is accessible on GitHub repository[8].

[8] https://github.com/kjerabek/Firefox-traffic-capture-DOH-enabled

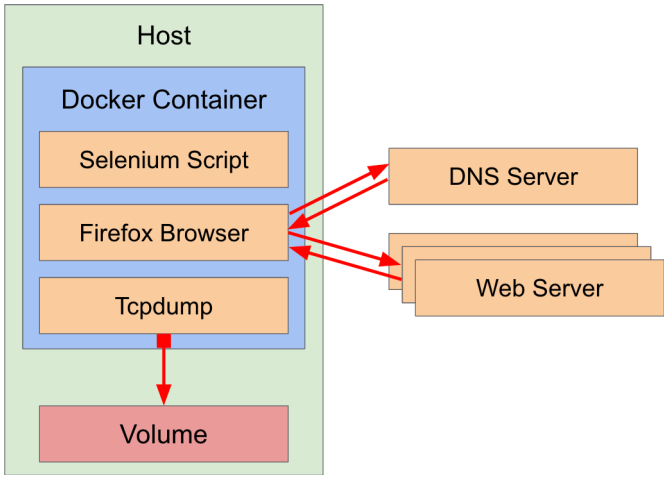


Fig. 11: Container structure for browser data generation.

5 Processing

The whole dataset processing, analysis, and training of a machine learning model were done on the big data cluster. The big data cluster and its management are based on container technology. The Docker containers are used as a base building block together with Kubernetes as cluster management.

The core of this system is a general big data framework Apache Spark. The Spark is deployed together with the Hadoop distributed file system (HDFS) that serves as storage for the source (unprocessed) data. The Apache Cassandra database then creates the storage for processed data.

The data are pre-processed; each packet is stored in the Cassandra database. Thanks to the Sparks accelerated in-memory SQL queries, it is possible to analyze this vast amount of packets very fast. The cluster is described in FIT-TR-2019-04 [4] technical report together with performance measurements.

6 DOH Servers

The number of public DOH servers is growing every month, as well as different public implementations of various DOH tools such as clients, servers, and proxies based on different programming languages. Community around curl and its doh client on GitHub is trying to keep us in touch with the current state by providing such a list of tools together with an updated list of 50 publicly available DOH servers on their wiki page[1].

The DOH servers provide a variety of features such as ad pages filtering, malicious pages filtering, DNSSEC support, family filtering, as well as non-filtered DNS. Users can then chose by the features that are also described at the wiki page.

The table below provides eleven chosen DOH servers from the wiki page supporting various features and TLS versions. Those servers were also used to generate traffic for the DOH traffic classifier.

DOH URLs	**Description**
https://doh.centraleu.pi-dns.com/dns-query	Ad-blocking, DNSSEC
https://dns.google/dns-query	Full RFC 8484 support
https://cloudflare-dns.com/dns-query	Available via Tor
https://mozilla.cloudflare-dns.com/dns-query	Mozilla
https://doh.opendns.com/dns-query	Experimental, DNSSEC
https://odvr.nic.cz/doh	Experimental, Knot resolver
https://doh.dnslify.com/dns-query	Strict privacy policy
https://dns.dnsoverhttps.net/dns-query	Toy-server, runs DOH proxy
https://doh.ffmuc.net/dns-query	DNSSEC, No logging
https://doh-de.blahdns.com/dns-query	Ad-blocking, DNSSEC
https://dnsforge.de/dns-query	DNSSEC

Table 3: List of DOH server domains used for generation.[1]

7 Single Request Protocols Analysis and Comparison

This section provides analysis and comparison of the DOH, DOT, and pure DNS over UDP. The DOH provides multiple options and formats of transport inside the HTTPS. Hence, there are three categories, such as DOH using POST, DOH using GET, and not standardized DOH using the JSON.

All of the data comes from the generated dataset in this work described in section 4. Over 1.4 million requests were analyzed for each of the categories. The same domains so as requests and responses in the same order were used for this generation.

The main point of interest here is to evaluate the difference between those protocols in terms of how much more data are transferred, how much more overhead in terms of sent packets each method adds, and how fast the single queries are.

The first graph 12 depicts an average exchanged packet number in every single query. It also contains TCP handshake with TLS authentication that, together with TCP acknowledgments, creates the most significant overhead in terms of the number of packets. This behavior can be seen mainly in comparison to the pure DNS over UDP, wherein most of the time, only two packets are needed for the whole query.

The second graph 13 shows an average payload size of packets summarized for the whole flow. The DNS over TLS has less average payload size against the other options, and the primary reasons will be given probably by avoiding the

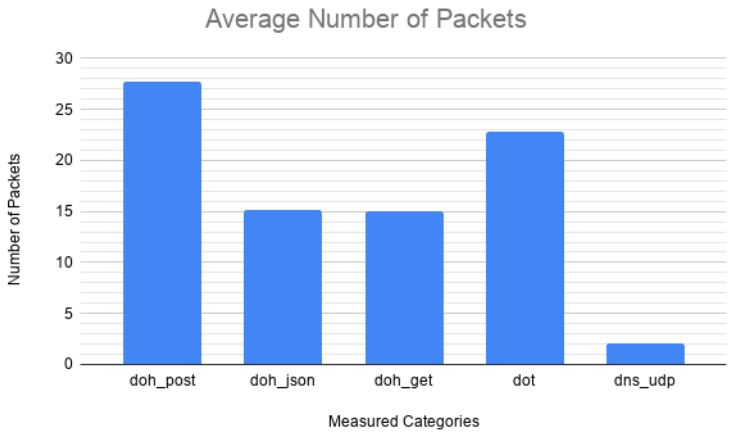


Fig. 12: Average packet number.

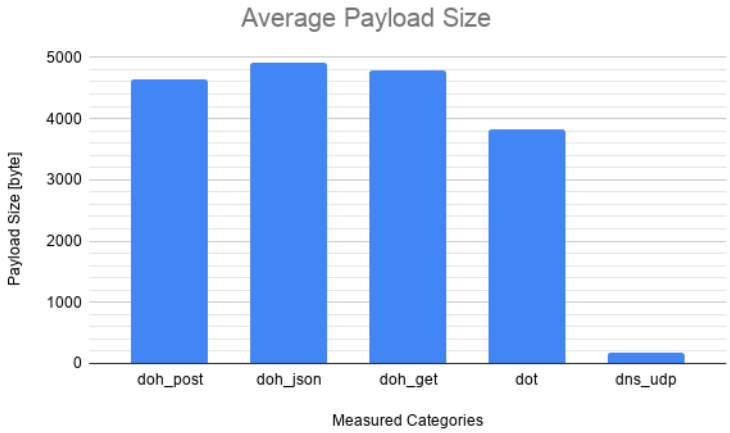


Fig. 13: Average payload size of the whole flow.

overhead caused by HTTP protocol. Pure DNS here represents much less payload size needed for the same information exchanged, but without any encryption.

The average duration of the whole flow is depicted in the graph 14. The length of the DNS over TLS is less than the other cases. Even the pure DNS that most of the time exchanges, only two packets have a higher duration. The second shorter option that can be observed is JSON transferred using HTTPS.

Surprisingly the pure DNS over UDP uses much less traffic in terms of packets and payload size. On the other hand, this method has a slightly worse perfor-

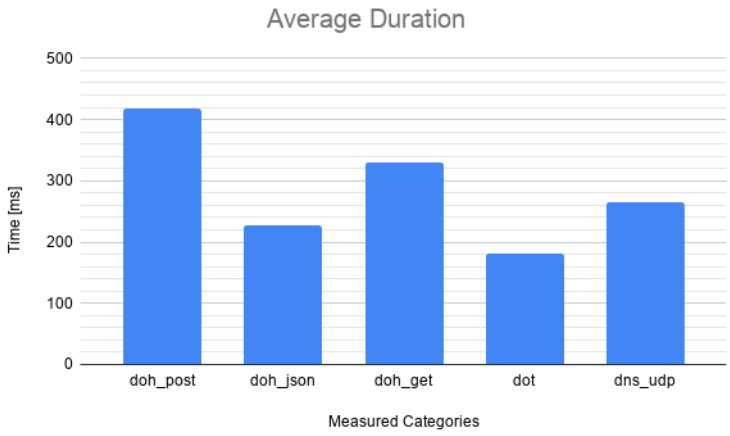


Fig. 14: Average duration of the whole flow.

mance than the others. The DNS over TLS overcomes the DNS over HTTPS options in those measurements.

8 Browser Behavior Analysis

This section provides deeper look into how browsers (the first applications currently incorporating this methods) are dealing with this new DOH standard.

8.1 Browsers DOH Enablement Options

We can categorize browsers by the different approaches they use into three groups. Those that are more closely tied to Chromium project[9] such as Google Chrome, Edge, Vivaldi, Opera. A completely separate group creates Firefox with its own implementation. Furthermore, Safari has not supported this feature yet.

Firefox Firefox provides the widest variety in terms of option tuning. User can enable DOH support in `network.trr` flags by choosing one of three modes, `off`, `first` (if first try not work, fall back to normal), `only` (use only trr mode). The `mode` is not working 100% times in terms of what user can expect, as only trr, but also fall back to normal according to the testing.

When users enable DOH, the default DOH resolver that points to the Cloudflare DOH server is used. This setting can be overridden to the user's desired DOH server. Nothing is dependent on the resolver set on the OS level. The method `POST` or `GET` for requests can also be chosen. Other options enable the

[9] https://www.chromium.org/

user to tune the behavior, such as setting the max number of retries, credentials, timeouts, bootstrap addresses, and so on.

Those multiple options for DOH tuning make Firefox browser a perfect testing behavior and traffic generator.

Google Chrome and the others Chromium has different behavior and provides only one option on how to tune DOH in the browser. Users can only enable or disable DOH. The browser then looks at the DNS resolver set on the OS level, and if the resolver is in a whitelist (table) of providers mapping IP addresses of approved DNS servers to their DOH URIs, then the browser will use it. The whitelist is of the fixed size currently, according to the web page provided by chromium developers[10]. Some of the servers are location dependent.

Chrome always looks at the DNS resolver set on the OS level, and if the server has DOH capable interface, it is on the whitelist. Then the DOH is used for the resolving. Otherwise, it is not used, and the regular DNS resolver is used instead.

This could be a better solution for non-experienced users, but the fixed whitelist could be an issue. Users on Linux are not allowed to enable the DOH feature at all.

Opera Nevertheless, Opera has chosen more strict approach. Users can enable DOH in the browser, but only one default resolver can be used (Cloudflare). There is no way how to change it to the user's server. Moreover, the VPN feature has to be disabled; otherwise, it is not working at all.

8.2 Loading Pages Impact

This performance test aims to reveal how big impact has the change from using classic DNS to DOH. At first, the DOH brings the overhead of TCP handshake together with a TLS handshake to the traffic at the beginning of each flow. Moreover, the traffic has to be encrypted and then unencrypted on both machines. Another disadvantage is that it adds more data in TLS headers and HTTP headers to DNS data itself. DOH needs more packets to be transferred, also because it deals with TCP traffic, and the acknowledgments have to be taken into account.

In this performance test, there were chosen $10,000$ website domains from the beginning of the majestic million list. The subsection 4.2 describes the data generation process. The performance test was run in the Docker environment three same servers in the same local area network at the same time. The conditions were the same for all categories. The $10,000$ page requests resolved by DNS, DOH POST, and DOH GET was repeated three times. The results are depicted in the following table 4.

The table shows averaged values from all three measurement runs. The outliers were cut off using the Z-score method. Three columns with results represent

[10] https://www.chromium.org/developers/dns-over-https

	domainLookupStart - domainLookupEnd	**domainLookupStart - responseEnd**	**domainLookupStart - domComplete**
	mean	**mean**	**mean**
DNS	**50.33**	**639.74**	3929.00
POST	65.13	652.94	**3795.50**
GET	71.68	660.57	3845.97

Table 4: Browser single page loading performance. In milliseconds.

the time from *domain lookup start* up to the end of some other action. The actions are called *navigation timing* and can be accessed in every browser[11]. The timers represent different time events from the beginning of navigation to the page.

The first column in the table represents time period from *domain lookup start* up to the current document *domain lookup end*. That should contain one request-response for one domain. The DNS is the fastest option here, as well as in the second case. The second column represents time period from the *domain lookup start* up to the current document *response end*. The *response end* timestamp is recorded as soon as the last byte of the current document is received. It is not surprising that the DNS the fastest option as this result in this column covers only one document and one domain lookup.

Whereas, the last column contain time period from *domain lookup start* up to the *dom complete*. The *dom complete* timestamp represents the time before the document readiness is set to complete. This should contain full web page loading, so multiple domain lookups, as well as files, downloading. Surprisingly, the DOH performs slightly faster than normal DNS.

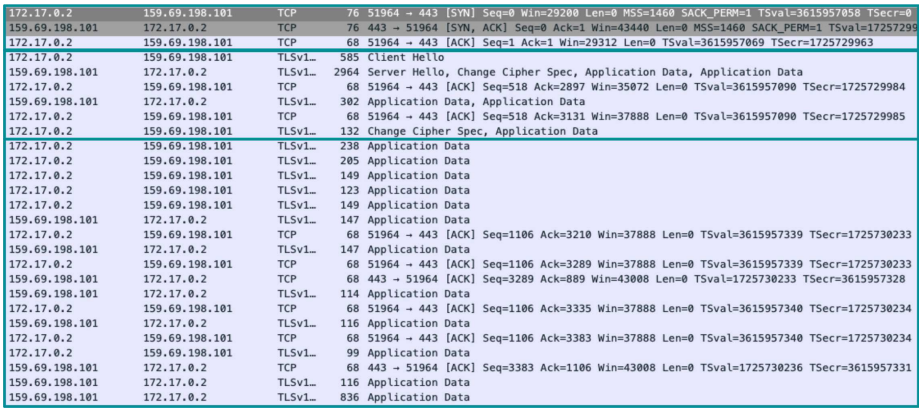
Parallelizing requests can cause this result once the secure connection is established with the DOH server. On the other hand, the regular DNS queries are transferred via UDP as separate requests. Could ISPs DNS monitoring tools cause the slowdown? The same Cloudflare servers were accessed in every case, so everything should be cached there, and the resolving itself should not impact this measurement.

8.3 Browser Traffic Analysis

The section 4.2 describes the process of single web page load traffic generation. The browser is always opened and closed again for each single web page access. Every cache is cleared, so the generated traffic and the behavior should be always the same. The whole analysis is primarily done using Firefox browser and later by Google Chrome.

[11] https://www.w3.org/TR/navigation-timing/

Firefox The Firefox browser with enabled DOH feature first resolves the domain of the DOH server and the certification authority domain to verify the certificate. When everything is successfully verified, the browser creates multiple connections towards the DOH server and sends requests. The number of created flows depends on the server and application. However, most of the time, one longer connection is maintained to gather all of the domains for a single web page request.



172.17.0.2	159.69.198.101	TCP	76	51964 → 443 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=3615957058 TSecr=0
159.69.198.101	172.17.0.2	TCP	76	443 → 51964 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM=1 TSval=17257299
172.17.0.2	159.69.198.101	TCP	68	51964 → 443 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=3615957069 TSecr=1725729963
172.17.0.2	159.69.198.101	TLSv1…	585	Client Hello
159.69.198.101	172.17.0.2	TLSv1…	2964	Server Hello, Change Cipher Spec, Application Data, Application Data
172.17.0.2	159.69.198.101	TCP	68	51964 → 443 [ACK] Seq=518 Ack=2897 Win=35072 Len=0 TSval=3615957090 TSecr=1725729984
159.69.198.101	172.17.0.2	TLSv1…	302	Application Data, Application Data
172.17.0.2	159.69.198.101	TCP	68	51964 → 443 [ACK] Seq=518 Ack=3131 Win=37888 Len=0 TSval=3615957090 TSecr=1725729985
172.17.0.2	159.69.198.101	TLSv1…	132	Change Cipher Spec, Application Data
172.17.0.2	159.69.198.101	TLSv1…	238	Application Data
172.17.0.2	159.69.198.101	TLSv1…	205	Application Data
172.17.0.2	159.69.198.101	TLSv1…	149	Application Data
172.17.0.2	159.69.198.101	TLSv1…	123	Application Data
172.17.0.2	159.69.198.101	TLSv1…	149	Application Data
159.69.198.101	172.17.0.2	TLSv1…	147	Application Data
172.17.0.2	159.69.198.101	TCP	68	51964 → 443 [ACK] Seq=1106 Ack=3210 Win=37888 Len=0 TSval=3615957339 TSecr=1725730233
159.69.198.101	172.17.0.2	TLSv1…	147	Application Data
172.17.0.2	159.69.198.101	TCP	68	51964 → 443 [ACK] Seq=1106 Ack=3289 Win=37888 Len=0 TSval=3615957339 TSecr=1725730233
159.69.198.101	172.17.0.2	TCP	68	443 → 51964 [ACK] Seq=3289 Ack=889 Win=43008 Len=0 TSval=1725730233 TSecr=3615957328
159.69.198.101	172.17.0.2	TLSv1…	114	Application Data
172.17.0.2	159.69.198.101	TCP	68	51964 → 443 [ACK] Seq=1106 Ack=3335 Win=37888 Len=0 TSval=3615957340 TSecr=1725730234
159.69.198.101	172.17.0.2	TLSv1…	116	Application Data
172.17.0.2	159.69.198.101	TCP	68	51964 → 443 [ACK] Seq=1106 Ack=3383 Win=37888 Len=0 TSval=3615957340 TSecr=1725730234
172.17.0.2	159.69.198.101	TLSv1…	99	Application Data
159.69.198.101	172.17.0.2	TCP	68	443 → 51964 [ACK] Seq=3383 Ack=1106 Win=43008 Len=0 TSval=1725730236 TSecr=3615957331
159.69.198.101	172.17.0.2	TLSv1…	116	Application Data
159.69.198.101	172.17.0.2	TLSv1…	836	Application Data

Fig. 15: Example of DOH traffic flow from Firefox browser.

One of the captured flows is depicted in figure 15. The flow can always be divided into three phases if we omit the connection finish. As the theory says, the first part is the TCP handshake. It is followed by the TLS handshake that can be of variable length (more or fewer packets) depending on information exchange and optional authentication. The data exchange phase follows immediately after those two handshakes. Multiple queries are sent to the DOH server; acknowledgment packets and responses follow those. The data sizes do not variate that much, as the same headers and the queries are almost the same lengths. The responses may differ as the server may send multiple records for one domain at once.

For open browsers, at least one longer DOH connection is expected. Firefox makes requests using the POST method by default. It also uses separate streams to send headers and requests.

Google Chrome Google chrome uses the GET method for queries. In addition, Chrome is not dividing headers and requests into separate streams; those are contained mostly in one packet.

9 DOH Identification

This section is focused on DOH identification in regular HTTPS. With the RFC 8484 that defines the DOH standard, the DNS requests are transferred as the payload of HTTPS. Because it is only a payload as such, it is similar to only another data transferred, it is hard even to tell if the user is using DOH or not. Since the traffic is hidden under the same port as regular HTTPS, it would be useful to have a clue that the user uses this protocol. It should bring some information to ISPs and enterprise network administrators. From quite a big captured and labeled traffic, it could be possible to make a broader analysis of the network flows that are the main subject of this work.

The topic focuses only on bi-directional network flows. Each flow is identified by source IP, destination IP, source port, destination port, and transport protocol. Only TCP flows are filtered by port 443 because the only aim of this work is to deal with HTTPS traffic, that is transferred using TCP.

The problem of DOH flows compared to the other HTTPS flows is that it could be very short or very long, depending on the application and user interactions. The flows can last even a couple of minutes or even longer. Hence, the collection of all packets or statistics from the whole flow should be less useful. Hence, the idea is to detect the DOH from only a few packets from the beginning of the flow.

Each flow starts with a TCP handshake, then the TLS handshake follows. Those have to be skipped. The TCP handshake is the same for every TCP flow so that we can omit it. The TLS handshake is more specific to the server itself and also depends on a client application. Both sides are providing multiple options for establishing secure connections together with an agreement and optionally authentication. That information can be extracted from TLS headers, and it can be useful for application or server identification, but completely useless for this particular problem.

Hence, the first and only useful information can be extracted from TLS application data packets. Those packets follow some behavior, it has a different size, and some patterns can be observed even only by looking at the captured network traffic.

The problem here is variability. Not only TLS protocol itself is currently widely used in multiple versions. Moreover, the TLS handshake phase may contain application data packets and other cipher change packets. This behavior makes the flow more variable.

The encapsulated protocol HTTP also brings more variability into the traffic. The protocol provides different features in all its versions; some are mentioned in 8.3. For example, a flow with encapsulated HTTPv2 can produce more or fewer packets based on if it uses multiple streams or not.

HTTP headers bring another variability. The DOH standard defined in RFC 8484 allows us to add multiple header fields. Adding more header fields than those necessary ones can extend the packet size.

9.1 Feature Identification and Selection

The features from network traffic have to be identified and selected to solve this classification problem. As discussed earlier, the information from TLS headers (mainly from TLS handshake) can help identify a particular connection, server, or client. The rest of the interesting information is encrypted. Hence, the idea is to speed up the process of parsing unnecessary information, and stay focused on the features that can be obtained up to the transport layer.

Those features should bring the necessary information to be able to identify the traffic. The features that can be taken into account are TCP flags, TCP payload size, TCP header size, packet direction, and timing. Those features can be extracted from each packet in the flow.

As was mentioned earlier, only a few packets from the beginning should be taken into account for the classification. The packets up to the end of the TLS handshake should be skipped. Based on the captured traffic from the Firefox browser and the analysis, we can say that in most cases, to skip the TLS handshake, 10 or 15 packets would be enough. Then, the next ten packets should be used for feature extraction and classification. Most of the flows can be classified using this method. Only small flows, containing up to 3 requests, can not fit this method, so we can omit them since the applications (mainly browsers) generates long connections always.

The best features observed during the testing were payload size, TCP push flag. The timings were not taken into account because it always depends on client and server distance, on the network segment, bandwidth, it also depends on the user interactions and so on.

9.2 Identification

Current state of the implementation in most browsers except for Firefox, allows users to enable DOH to only limited amount of specified servers. Hence, the easier approach currently is to maintain and detect the IP addresses of those servers.

On the other hand, the Firefox browser allows users to enable DOH towards all available, even private DOH servers and proxies. Moreover, if other applications and operating systems will enable the support of encrypted DNS using DOH, it should be necessary to make detection from the traffic itself.

The dataset used for training and testing ML method were generated using the process described in section 4.2. At first, the requests were made using GET and POST methods towards default Cloudflare DOH resolver. The second requests were made using GET and POST methods towards Google DOH resolver. The dataset were filtered for only HTTPS traffic and later filtered and labeled by the IP address. The DOH packets had the IP of DOH resolver and the rest of the traffic belonged to regular HTTPS.

Random Forest algorithm was chosen out of other machine learning algorithms for its good features among the other methods. The other methods taken into account was Naive Bayes and Multi-layer Perceptron Neural Network. The

algorithm selection was limited only to algorithms provided by the Apache Spark Machine Learning library.

Multiple combination of data from dataset were tested. If we consider a model trained using dataset with DOH and normal traffic flows and tested on different part of this dataset, the accuracy of identification was 99.9%. The features used for the classification was push flag, TCP header size and TCP payload size from second ten packets. The amount of data needed for training model that achieved such accuracy was $8,000$ of DOH GET flows and $8,000$ of DOH POST flows queried towards the same DOH server with different domain in query. This $16,000$ packet flows was combined with $16,000$ flows of normal HTTPS traffic to make balanced training dataset for the model. Then the model was tested on different part of the dataset with previously mentioned 99.9% accuracy. But such a model was not able to classify DOH towards different DOH server with higher than 47% accuracy.

New dataset was generated with some changes based on the knowledge from the previous experimentation described in paragraph above. The new dataset contained 100 requests for each different method such as GET, POST, and both of those methods with additional HTTP headers. Each of those methods were queried towards 11 different servers randomly chosen from currently available DOH servers providing different features. The servers that was used as resolvers are described in table 3.

The accuracy of 96.7% was achieved using this a bit more variable training set. Even there was less flows in the training set. The trained model was tested towards different DOH server with such success. The input parameters were tested by cross validation method provided by Spark Machine Learning library and the best result using random forest classifier was achieved using max bins set to 90, maximal depth set to 40 and number of trees set to 18. This research should be extended and more tests should be done.

10 Conclusion

This technical report describes the encryption of DNS protocol by new standard DNS over HTTPS, together with its alternatives and its predecessors. It provides an analysis of the DOH protocol and also a comparison to the others.

Together with protocol analysis, the performance analysis of the protocol and impact on browsers is part of this report. The browser used for testing in this report is Firefox. Firefox was chosen because of its impact on the standard, it was the first browser that supported this standard, and it provides the best configuration options for tuning the usage of DOH.

The performance of the protocol was also compared to other protocols. This was tested using simple open-source command-line tools currently available that are able to resolve DNS queries using those protocols. The DOH GET, DOH POST, DOH JSON DOT, and DNS over UDP was compared using single requests towards the same DNS resolver (Cloudflare) queried at the same time from different machines located on the same local area network. The best average duration was surprisingly achieved using DOT and DOH JSON, followed by DNS over UDP. Surprisingly the DNS over UDP was a bit slower even though most of the requests contain only two packets compared to the other protocols that have to do TCP and TLS handshake and finish the connection in advance.

The browser loading pages performance analysis revealed that there is little difference in time for loading pages, whether the DNS over UDP, DOH POST, or DOH GET is used. However, the fastest option was using the DOH POST method.

The last part of the work focuses on the analysis of browser behavior. Because browsers are currently the only advanced applications implementing and using DOH, the work focuses on identifying DOH in regular HTTPS. The experimentation with identification itself is done in Apache Spark using the Spark Machine Learning library. The Random Forrest classifier was chosen for training the model and testing. The best result of accuracy 96.7% was achieved using features extracted from the second ten packets in each HTTPS flow. The result is preliminary, and it was not tested deeply if the model does not fall into underfitting or overfitting.

This work was done within the TARZAN project [10].

References

[1] Curl Authors, ed. *DNS over HTTPS*. 2020. URL: https://github.com/curl/curl/wiki/DNS-over-HTTPS (visited on 05/21/2020).

[2] Tim Dierks and Eric Rescorla. "RFC 5246-the transport layer security (TLS) protocol version 1.2". In: *Internet Engineering Task Force* (2008).

[3] Z Hu et al. *RFC 7858-Specification for DNS over Transport Layer Security (TLS)*. 2016.

[4] Kamil Jeřábek. *Container Based Big Data Cluster: Technical report*. Tech. rep. 2019.

[5] Matt Larson et al. "DNS security introduction and requirements". In: (2005).

[6] Majestic-12 Ltd, ed. *The Majestic Million*. 2019. URL: https://majestic.com/reports/majestic-million (visited on 12/10/2019).

[7] Patrick McManus and Paul Hoffman. "DNS Queries over HTTPS (DoH)". In: (2018).

[8] Paul Mockapetris. "Domain Names—Concepts and Facilities, RFC 1034". In: *ISI, November* (1987).

[9] Paul Mockapetris et al. "RFC-1035 Domain Names-Implementation and Specification". In: *Network Working Group* 55 (1987).

[10] Matoušek Petr et al., eds. *Integrated platform for analysis of digital data from security incidents*. 2019. (Visited on 09/10/2019).

[11] Tirumaleswar Reddy, Prashanth Patil, and Dan Wing. "Dns over datagram transport layer security (dtls)". In: (2017).