A HANDS-ON GAZE ON HTTP/3 SECURITY THROUGH THE LENS OF HTTP/2 AND A PUBLIC DATASET

A PREPRINT

Efstratios Chatzoglou
Dept. of Information and Comm. Systems Engineering
University of the Aegean
Karlovasi 83200, Samos, Greece
efchatzoglou@gmail.com

Vasileios Kouliaridis
Dept. of Information and Comm. Systems Engineering
University of the Aegean
Karlovasi 83200, Samos, Greece
bkouliaridis@aegean.gr

Georgios Kambourakis *
European Commission
Joint Research Centre
Ispra 21027, Italy
georgios.kampourakis@ec.europa.eu

Georgios Karopoulos
European Commission
Joint Research Centre
Ispra 21027, Italy
georgios.karopoulos@ec.europa.eu

Stefanos Gritzalis
Department of Digital Systems
University of Piraeus
Piraeus 18532, Greece
sgritz@unipi.gr

August 16, 2022

ABSTRACT

Following QUIC protocol ratification on May 2021, the third major version of the Hypertext Transfer Protocol, namely HTTP/3, was published around one year later in RFC 9114. In light of these consequential advancements, the current work aspires to provide a full-blown coverage of the following issues, which to our knowledge have received feeble or no attention in the literature so far. First, we provide a complete review of attacks against HTTP/2, and elaborate on if and in which way they can be migrated to HTTP/3. Second, through the creation of a testbed comprising the at present six most popular HTTP/3-enabled servers, we examine the effectiveness of a quartet of attacks, either stemming directly from the HTTP/2 relevant literature or being entirely new. This scrutiny led to the assignment of at least one CVE ID with a critical base score by MITRE. No less important, by capitalizing on a realistic, abundant in devices testbed, we compiled a voluminous, labeled corpus containing traces of ten diverse attacks against HTTP and QUIC services. An initial evaluation of the dataset mainly by means of machine learning techniques is included as well. Given that the 30 GB dataset is made available in both pcap and CSV formats, forthcoming research can easily take advantage of any subset of features, contingent upon the specific network topology and configuration.

Keywords HTTP/2 · HTTP/3 · QUIC · IDS · Machine Learning · Anomaly Detection · Vulnerabilities · DDoS · Attack · Dataset.

*Corresponding author

1 Introduction

HTTP was originally designed without focusing on security and reliability; this is one of the main motivations behind the development of HTTP/2 [1]. However, as we discuss in detail in Section 2, the adoption of HTTP/2 introduced new attacks, as happened also in the past with the rather quick release of novel technologies that were later found to have security issues [2, 3, 4, 5]. The next major HTTP version, namely HTTP/3 [6], is an upgrade of HTTP/2 in terms of performance, reliability, and security; at the same time, it is based on the QUIC protocol [7] and it heavily changes the way web browsers and servers communicate, given that it uses UDP as a transport layer protocol instead of TCP, making it a candidate source of further security issues. Considering also stagnating security issues of HTTP, such as the low penetration rate of HTTP security headers [8] (below 17% across all platforms), as well as VNC [9, 10] and iframe [9, 11] phishing attacks, the following question arises: what is the security status of the new generations of HTTP, that is, HTTP/2 and HTTP/3?

Deployment-wise, according to [12], HTTP/2 has currently an adoption rate of 45.2%, which is at about the same level as one year ago (45.4%). In between, this rate went up to a maximum of 46.9% in Jan. 2022, following a declining trajectory ever since. HTTP/3, on the other hand, followed a steady upward adoption rate from 19.5% one year ago to 25% in Jun. 2022. It is also noteworthy that a new candidate was added to the existing ones for supporting encrypted DNS [13], that is, DNS over HTTP/3 or DoH3 [14]. These data indicate that the adoption of HTTP/2 is relatively stable, but losing ground and HTTP/3 is taking its place, albeit in a slow pace. This underlines the need to evaluate the security of HTTP/2, with a view to protect today's vulnerable deployments, but at the same time consider the issues that HTTP/3 will bring in the near future when it will become the dominant protocol version.

Even though a significant mass of work has been accomplished on the analysis of HTTP/2 vulnerabilities, to the best of our knowledge, no extensive review exists to provide a spherical analysis on the security of HTTP/2. In fact, existing works in this field investigate individual attacks on HTTP/2, whereas very few of them evaluate HTTP/2 against a wide variety of attacks. Moreover, no insight is provided into the applicability of HTTP/2 attacks to the latest HTTP/3 version. The work at hand aims to address the aforementioned issues and provides the following contributions:

- A comprehensive review of HTTP/2 security and known attacks in the literature.
- A discussion on which HTTP/2 security attacks could be applicable to HTTP/3 as well.
- A hands-on evaluation of QUIC and/or HTTP/3 enabled servers against HTTP/2 and HTTP/3 attacks.
- A state-of-the-art dataset built to evaluate HTTP/2, HTTP/3, and QUIC security, as well as a thorough evaluation of the proposed dataset, mainly by means of machine learning techniques.

The paper is organized as follows. The next section surveys several types of attacks on HTTP/2 and discusses their portability to HTTP/3. Section 3 provides an evaluation of QUIC and/or HTTP/3 enabled servers against common attacks. In Section 4, we present our new dataset, created specifically to assess the security of the latest HTTP protocols. Section 5 is devoted to the evaluation of the proposed dataset. The last section concludes.

2 Categories of attacks against HTTP/2

This section surveys the major categories of attacks against HTTP/2; moreover, the discussion focuses on if and to what degree a specific category migrates to HTTP/3. It should be noted here that previous work on web attacks [15, 16, 17, 18, 19, 20, 21] has shown that server implementations are exposed to issues such as URL parsing, which may lead to server-side request forgery (SSRF) or path traversal attacks, and cache poisoning, which can enable an opponent to steal information or mount a remote code execution (RCE) attack. Additionally, works such as [22, 23], illustrated different empirical attacks based on TLS vulnerabilities that could lead to MitM attacks. While the aforementioned assaults concern server-side attacks over the HTTP, they are irrelevant of the HTTP protocol version used and they are considered to be out-of-scope of this paper; thus, such attacks are omitted from the analysis that follows.

2.1 Amplification attacks

The work in [24] examined the possibility of amplification attacks, termed HTTP/2 Tsunami, by capitalizing on the HTTP/2's HPACK header compression method. To store the requested headers in a first-in first-out fashion [25], HPACK uses a dynamic table. The authors assumed that, by exploiting HPACK, HTTP/2-enabled proxies could be used as amplifiers. To this end, they calculated the exact length of each packet header based on the length of the dynamic table, which can be assigned by the `SETTINGS_HEADER_TABLE_SIZE` field, having a default value 4 KB. They simultaneously sent multiple packets to the Nginx and nghttp2 proxies, with the assist of three different headers, namely, *Authority*, *User agent*, and *Cookie*. They resulted into having four cases with a bandwidth amplification factor

of 79.2, 94.4, 140.6, and 196.3, for 100, 128, 256, and 512 maximum concurrent requests, respectively. The latter field is directly related to the dynamic table and refers to the number of simultaneous connections the server can handle at one time. The authors mentioned that they altered this field for each assault, given that this field is directly related with the amplification factor. It should be noted that the 100 max concurrent requests was the default value on each proxy.

Regarding HTTP/3, in the recently published RFC [6], HPACK was replaced by QPACK, due to the incapability of QUIC to handle an order (first-in first-out). QPACK handles requests differently, thus, the HTTP/2 Tsunami attack is not directly applicable against HTTP/3 proxies. Nevertheless, it is interesting to examine whether the main ideas behind this attack could affect HTTP/3.

2.2 Cryptojacking attacks

The authors in [26] explored the feasibility of taking advantage of HTTP/2 proxies to perform cryptojacking, that is, consuming resources for mining cryptocurrencies without the consent of the resources' owner. Precisely, the attacker has access to an HTTP/2 proxy which is orchestrating the attack with the assist of the `mitmproxy` tool. First, the victim requests to visit a specific domain through the proxy. In turn, the malicious proxy requests over HTTP/1.1 the corresponding content from the respective web server. The latter responds with an Upgrade header (101), changing the connection to HTTP/2 over cleartext (h2c). The malicious proxy accepts that request, receives the data from the web server, and injects a cryptojacking payload in the form of Javascript code. Finally, the victim's machine receives the web content and executes the cryptojacking code, starting unwillingly the cryptomining procedure. Regarding mitigation methods, the authors suggested that such attacks can be blocked by any adblock software; on the other hand, such blocking could potentially be evaded by encrypting the cryptomining Javascript code with the assist of a custom stratum pool [27].

Concerning the portability of cryptojacking to HTTP/3, we argue that this attack is based on modifying the connection to a cleartext one; given that HTTP/3 does not have a cleartext mode, the attack cannot be applied as is. However, it is interesting to note here that the execution of this attack, as presented in [26], is questionable; RFC 7230 [28] states that "*A server must not switch to a protocol that was not indicated by the client in the corresponding request's Upgrade header field*". In other words, a server would never initiate a protocol upgrade, but it would do so only after a client sent an upgrade request.

2.3 Denial of Service attacks

The work in [29] presented a DDoS attack model where malicious traffic mimics flash crowds, based on the assumption that legitimate HTTP/2 flash crowd traffic has the same network characteristics as a distributed denial-of-service (DDoS). Specifically, they investigated four different cases: (a) a flood-based DoS, (b) modifying the `WINDOW_UPDATE` size, (c) modifying the number of packets, and (d) finding the minimum number of attacking bots to mount a successful DDoS attack using the parameters found in the previous two cases. The results showed that HTTP/2 does not limit the exchanged traffic, and additional mechanisms should be devised to monitor and react to network patterns that could lead to DoS. This work is based on a similar testbed setup as [30], whereas both are based on a known vulnerability on flow control and more specifically on the `WINDOW_UPDATE` size, which has been identified as a potential waste of resources if abused [1]. Given that [30] examined slow rate DoS attacks, it is further analyzed in Section 2.4. Notably, attacks that are related to `WINDOW_UPDATE` are infeasible on HTTP/3 because that field was removed from the specification [6].

The work in [31] presented an experimental analysis of the vulnerability of HTTP/1 and HTTP/2 against flood DDoS attacks. The authors created an experimental setup comprising two Linux hosts linked with a 1Gbit Ethernet link; one of the hosts was the web server, based on nghttp2 v1.10.1, and the other was used to flood the server with requests. The scenario involved generating the maximum number of requests possible, using 800 simultaneous connections from the client, first against an HTTP/1 and then against an HTTP/2 server. The HTTP/2 RFC [1] recommends the `Max_Concurrent_Requests` value to be set to 100 on the server; the authors have used different values, from 100 to 512, to evaluate how this parameter affects the assault. The results showed that, in both cases, the bottleneck of the attack was the packet generation on the attacker side due to limited processing power and offload capability of the network card. The main difference between the two experiments is that, due to multiplexing, 57 times more packets were created and sent in HTTP/2 with the recommended `Max_Concurrent_Requests` value of 100; on the other extreme, with this value set to 512, the potency of the attacker rose to 95 times more packets compared to those sent in HTTP/1. This suggests that, even though HTTP/2 provides some performance benefits, it makes flood attacks more effective at the same time. Overall, this attack is a typical HTTP flooding attack, exploiting HTTP/2 characteristics to become more effective; under this prism, it is possible that a similar attack can affect HTTP/3 as well.

The work in [32] examined six different attacks that could theoretically affect a 5G core network using HTTP/2 as an application layer protocol for service-based interfaces. This work is purely theoretical and does not provide any

implementation of the suggested attacks. Moreover, even though not explicitly mentioned, these assaults can be launched only in a 5G network by an insider opponent who has access to the core network. In the following, we describe the four DoS-related attacks, whereas the two remaining privacy-related attacks are analyzed in Section 2.6:

1. Stream reuse attack: In a 5G network, one Network Function (NF) can establish multiple connections to another NF, whereas an HTTP/2 request/response utilizes a single stream. According to RFC 9113 [1], "*The identifier of a newly established stream must be numerically greater than all streams that the initiating endpoint has opened or reserved*". Therefore, each stream ID can only be used once and when all IDs have been exhausted, the NF should establish a new connection to the other NF. In this context, an attacker could impersonate an NF, causing stream ID and connection exhaustion to legitimate NFs. Considering that the recommended `Max_Concurrent_Requests` in HTTP/2 are 100, having a finite number of stream IDs can be fatal for the core 5G infrastructure. Additionally, the reuse of an already used stream ID for a new stream could make the server crash.
2. Flow control attack: Similar to the previous one, this attack exploits the multiplexing capabilities of HTTP/2. In this case, the attacker requests a large resource and at the same time sets a very small `WINDOW_UPDATE` size. This way, the server is forced to send the data in a slow pace in many different streams, while consuming resources to process these streams. By launching multiple such requests, it is possible to render the server unable to process further requests.
3. Dependency and priority attack: HTTP/2 provides a priority mechanism, used to process higher priority requests before lower priority ones. Prioritization in stream multiplexing is further supported by a dependency tree, which is a graph that stores dependencies among streams, for example, stream "A" should be completed before stream "B" starts. However, the size of the tree is not limited, and an NF could be tricked into creating a dependency tree that consumes its memory by, for example, creating infinite loops.
4. Header compression attack: It is based on the HPACK compression mechanism used in HTTP/2. A scenario that could lead to a DoS attack in this case is the creation of an "*HPACK bomb*": an attacker creates a special compressed message, which forces the targeted machine to use a large amount of memory after its decompression.

From the above-mentioned attacks, only the stream reuse could be possible against HTTP/3. The flow control and the dependency and priority attacks cannot be exploited in HTTP/3, as stream-level multiplexing is provided by QUIC. Similarly, the header compression attack is inapplicable to HTTP/3 as the HPACK mechanism has been replaced by QPACK. Furthermore, it mainly depends on the existence of a zero-day vulnerability on the attacked endpoint, which is irrelevant of the HTTP version used.

The authors of [33] proposed the H_2DoS attack, a novel application-layer DoS attack against HTTP/2 that exploits its multiplexing and flow control mechanisms. Specifically, they capitalized on two different frame types of HTTP/2 that play an important role in flow control, namely, `SETTINGS` and `WINDOW_UPDATE`. Similarly to previous attacks, H_2DoS exhausts server resources by initiating and maintaining active a huge number of HTTP/2 connections. The authors demonstrated that their attack could lead to a DoS by occupying all available connections; they also compared their attack against two other well-known DoS assaults, namely, *slowloris* and *thc-ssl-dos*. The results showed that H_2DoS was more effective in comparison to the other two, raising both CPU and memory usage to $\approx$40% and 10%, respectively. On the other hand, their comparison showed that slowloris consumed more CPU after 12 min, having an average of 50% CPU usage, whereas H_2DoS dropped to 30%. Regarding the repeatability of this attack, it should be noted that not all the necessary information, such as field values, are available. Again, as with other similar attacks, H_2DoS is infeasible in HTTP/3, as the above-mentioned flow control fields were removed.

The work in [34] proposed a new DDoS attack, dubbed Multiplexed Asymmetric attack, where computationally intensive requests are multiplexed together. The main scenario tested by the authors was sending multiple requests to cause CPU exhaustion to the HTTP server. On top of this application layer attack, if the server supported Server Push, a flooding DDoS attack was triggered at the network layer. The Server Push feature used in this last case is responsible to preemptively deliver data packets to the client before even requesting them. Both HTTP/1.1 and HTTP/2 servers were tested against the Multiplexed Asymmetric attack under the same load, and the results showed that the HTTP/2 version was more resilient. Also in this case, the necessary information to reproduce the attack, such as the attack scripts, are not available. Regarding the migration of the attack to the latest HTTP version, while HTTP/3 supports multiplexing and Server Push, they are implemented with different mechanisms, making these attacks not directly applicable.

2.4 Slow rate attacks

Even though slow rate attacks are essentially a subcategory of DoS attacks, we chose to present them separately due to their stealthier nature that requires more effort and different detection methods. In [30], a DoS attack variant was

introduced, which is based on sending low-rate traffic that contains resource-hungry instructions to a victim HTTP/2 server. This work takes advantage of the same flow control vulnerability that manipulates the WINDOW_UPDATE size as in [29], which has been analyzed in Section 2.3. The authors, using a custom testbed, answer three main questions: (a) how DoS attacks towards an HTTP/2-enabled server can be mounted, (b) how many servers can a single client instance attack successfully, and (c) how can attacks be stealthier by introducing time delays in the attack traffic. The experimental evaluation involved five different test cases, all of which had an increased CPU usage between 88% and 98%, showing that a DoS attack is feasible. Regarding (b), an attacker with a single client was able to assault successfully 12 server machines and, thus, no additional attacking resources to interrupt HTTP/2 services are needed such as, for example, distributed machines in DDoS attacks.

Finally, the introduction of time delays from 1 to 100 ns did not make the attack stealthier, suggesting that slow rate attacks against HTTP/2 are impracticable. Nevertheless, we argue that the delay of 1 to 100 ns is too short to consider the attack a slow rate one. For instance, the analysis in [35] demonstrated that a slowloris assault needed ≈5,882 packets per second on an HTTP connection, whereas in the current attack with the lowest-rate scenario (one packet every 100ns) 10 million packets were sent in the same duration, resulting in 1,700 times more packets. Given that WINDOW_UPDATE has been removed in HTTP/3, this attack is not directly applicable.

The authors of [36] proposed zAttack, a new slow rate DoS attack that exploits the invalid frame state vulnerability of HTTP/2. Precisely, the attacker sets the SETTINGS_MAX_CONCURRENT_STREAMS field to 0 to indicate that the server cannot create new streams, apart from a stream with ID 0 for exchanging configuration data and another one with ID 1 for exchanging request data. In the next step, the server acknowledges the configuration sent by the client, as well as sends its own negotiation parameters. Normally, the client acknowledges the server parameters and the data exchange starts. During zAttack though, the attacker sends an RST_STREAM frame that closes the stream with ID 1, leaving the stream with ID 0 open. Since the server cannot create a new stream to send data, it waits for a specific time and, if the client does not create a new stream, it drops this open stream. A large number of such open streams, however, can render the server unable to serve additional requests, resulting in a DoS. It should be mentioned at this point that it is not clear from the authors why closing the stream with ID 1 is necessary, when keeping both streams open could actually assist in consuming server resources faster. The authors tested their attack against three different web servers, namely, Apache2 v2.4.33, Nginx v1.14.0, and H2O v2.3.0. The results show that each server had a different timeout period, i.e., 60, 300, and 10 secs, for Apache2, Nginx, and H2O, respectively. It was also observed that the maximum number of simultaneous connections that each server could handle was 400, 1024 and 2030, respectively. The required rates to bring down the servers are 6.7, 3.4, and 203 requests/sec; these data suggest that in the H2O case the attack could be more easily detected. Regarding HTTP/3, it is not possible to mount the zAttack given that the SETTINGS_MAX_CONCURRENT_STREAMS field was removed in the latest HTTP version [6].

The authors of [37] examined slow rate DoS attacks in HTTP/2 and proposed an anomaly-based detection method. Specifically, they presented how slow rate DoS attacks can consume a web server's connection pool by injecting specially crafted HTTP requests. The authors tested their proposals using popular web servers, namely, Nginx 1.10.1, Apache 2.4.23, Nghttp2 1.14.0, and H2O 2.0.4, in their default settings and the results showed that most of them are vulnerable to the proposed assaults. The testbed includes a server running Kali 2.0, which hosts the webservers, as well as two client computers, a malicious and a genuine, both running Ubuntu 16.04 LTS. The malicious client was used to launch slow rate DoS attacks, while the genuine client was used to check the server's availability during these attacks. In more detail, the authors implemented five different attacks:

1. In the first attack, the malicious client sends an HTTP/2 payload with a SETTINGS frame with SETTINGS_INITIAL_WINDOW_SIZE field equal to zero, as well as a complete GET request. When this field is set to zero, the server assumes that the client cannot receive any data right now and waits for WINDOW_UPDATE frames from the client. The malicious client never sends WINDOW_UPDATE frames to the server, which makes the server wait for a while, depending on its configuration. The authors found that Nginx and Nghttp2 waited for 60 sec, Apache for 300 sec, and H2O waited indefinitely.
2. Similarly, in the second attack, the malicious client sets and resets the END_HEADERS and END_STREAM flags of the HEADERS frame respectively, and then sends a complete POST request. The server assumes that one or more DATA frames are yet to be received, due to the END_STREAM flag reset. Nghttp2 waited for a maximum of 975 sec, while Apache, Nginx, and H2O waited indefinitely on repeated attacks.
3. In this attack, the malicious client sends a Connection Preface message to the server after the connection establishment; this makes the server wait to receive an HTTP request that is never sent. Nginx waited indefinitely on repeated attacks, while Apache, H2O, and Nghttp2 waited for 300, 10, and 975 sec, respectively.
4. In the next slow rate DoS attack, there are two flavors: the malicious client sends a HEADERS frame with END_HEADERS and END_STREAM flag reset and set, respectively, or both flags reset. The server assumes that it received an incomplete header and waits to receive the complete header block, which is never sent. When

this attacks is repeated, Apache and H2O wait indefinitely, while Nginx and Nghttp2 wait for 90 and 60 sec, respectively.

5. In the last attack scenario, the malicious client sends a GET or POST request. When the server responds to this request, it sends a `DATA` frame along with two `SETTINGS` frames. Normally, the second `SETTINGS` frame must be acknowledged by the client; however, the malicious client never sends back an acknowledgement. As a result, vulnerable web servers wait for some time before closing the connection. Apache, Nginx, H2O, and Nghttp2 waited for 5, 180, 10, and 975 sec, respectively.

As a defensive measure, the authors proposed an anomaly-based technique to detect these types of attacks, which works by comparing observed traffic with expected patterns. Their method was able to detect these attacks with high accuracy. The first attack is infeasible on HTTP/3, given that the `WINDOW_UPDATE` field was removed from the specification. For the `END_STREAM` issues, i.e., attacks 2 and 4, RFC 9114 [6] defines that this field is optional, since QUIC is responsible for handing stream traffic. As a result, such an attack will be possible only if the HTTP/3 implementation uses this field. Attack 3 is not possible on HTTP/3 since the Connection Preface message is not part of the specification. On the other hand, HTTP/3 still implements control frames, i.e., `SETTINGS`; thus, it is possible that the 5th attack is still applicable to HTTP/3.

2.5 HTTP/2 smuggling attacks

In [38], the port of HTTP request smuggling to HTTP/2 is investigated. The author exploited the Upgrade header (101) to upgrade HTTP/1.1 connections to HTTP/2 over cleartext (h2c), while having a reverse proxy as an intermediate. The result of such an attack is that a malicious client can establish unrestricted HTTP connections with back-end servers. This way an attacker is able to bypass reverse proxy access controls or restrictions such as accessing a directory. Although this attack is considered a misconfiguration, the author suggests blocking Upgrade requests or limit them only to the necessary services (e.g., `websocket`). Given that HTTP/3 does not have a cleartext mode, this attack does not apply to it.

The work in [39] illustrated different techniques against web applications to create an HTTP request smuggling attack, the Achilles heel of HTTP/1.1 protocol. While the issues mentioned were patched by the respective website owners, similar techniques can possibly affect other web implementations due to different HTTP/2 misconfigurations. The author presented the following three HTTP/2 desync scenarios, in which an HTTP request smuggling attack was feasible through an HTTP/2 connection:

1. HTTP/2 desync attack: Such an attack can occur when a front-end server communicates with clients on HTTTP/2, but uses HTTP/1.1 to communicate with the back-end server. The main cause of such attacks is that the front- and back-end cannot agree on which of the Content-Length or Transfer-Encoding header to use for obtaining the request length. This type of attacks can allow an attacker to inject arbitrary prefixes to HTTP requests of other users, steal passwords and credit card numbers, or even make the front-end send the response intended for a user to a different user.
2. Desync-powered request tunnelling: This is a subclass of the previous attack, and it relies on the connection-reuse strategy followed by the front-end. When a request arrives at the front-end, it has to decide whether it will forward it using an already established connection with the back-end or create a new one; this decision affects the possible attacks that can be mounted. The range of assaults includes requests reaching the back-end without being processed by the front-end, exploiting internal headers injected by the front-end, and web cache poisoning.
3. HTTP/2 exploit primitives: Different exploit techniques were illustrated against HTTP/2 in this case. For instance, it is possible to send requests with multiple methods or paths, lead to server-side request forgery (SSRF), enable request line injection which allows bypassing block rules in the back-end server, and tamper with internal and external headers.

While these assaults were identified mostly against web applications, it is possible that they could also be used against other HTTP/2-based connections. Furthermore, even though RFC 7540 protects from such methods (for example, the transfer encoding header field is forbidden in HTTP/2) some servers accepted it. For this reason, it is possible to affect HTTP/3-enabled servers as well.

2.6 Privacy attacks

Suresh et al. [40] provided an overview of the HTTP/2 protocol and a short discussion on security issues, such as Head-of-Line Blocking and DoS attacks. The main objective of this work was to investigate the feasibility of

decrypting HTTP/2 traffic, using a suitable HTTP/2 test environment. The authors found out that by exploiting the SSLKEYLOGGING feature, i.e., a mechanism used by browsers to log private keys into a file, it is possible for an attacker to extract private information, such as websites visited, Operating System (OS), and browser version. However, the authors neither provided a complete survey on existing HTTP/2 attacks nor examined the possibility of their migration to HTTP/3. Since this issue pertains to TLS decryption, it is considered pertinent to HTTP/3.

In [32], a MitM attack against a 5G network using HTTP/2 for service-based interfaces is presented. In this assault, the attacker first performs a DNS poisoning attack to insert a malicious NF between two legitimate NFs. Then, the intercepted traffic can be snooped even if decryption of the TLS traffic is needed, similar to [41, 23]. Another family of privacy attacks that can be mounted in the same setting is interconnection attacks. Opponents can track users or eavesdrop private information when different networks interconnect if the existing security mechanisms are misconfigured or not deployed at all. To succeed in the aforementioned attacks, the attacker should have access to the 5G core network. Regarding portability to HTTP/3, MitM and interconnection attacks are irrelevant of the HTTP protocol, that is, they could be possible either with or without the existence of HTTP/2 or HTTP/3 in the absence of proper security mechanisms.

The authors of [42] compared the resilience of both HTTP/1.1 and HTTP/2 against state-of-the-art web fingerprinting attacks. Specifically, they collected the 99 most popular websites as ranked by Alexa [43] to get the dependency structure of each site as well as the size of each site's resources. The Chrome DevTools protocol was used to log requests and responses sent or received by the browser, as well as intercept network events, such as `requestWillBeSent`, `responseReceived`, `dataReceived`, and `loadingFinished`. The authors used this information to create models of the features that influence the network trace of loading a page. These models were then served on both HTTP/1.1 and HTTP/2, using the Caddy web server to compare their susceptibility to fingerprinting. Additionally, the *tcpdump* tool was used to export network traffic to pcap files. These files were then processed to filter out DNS packets, and clear out TCP packets with no data, recording only the direction, size, and timing of each packet. These attributes were used as input in a random forest model to perform fingerprinting. According to the results, the model in HTTP/1.1 achieved an accuracy of 80%-99%, while in HTTP/2 with server push enabled the accuracy diminished to 74.2%, showing a smaller attack surface. According to [44], QUIC protocol can evade up to 96% of TCP-trained classifiers; however, they conclude that QUIC shows a similar difficulty of fingerprinting as TCP.

The work in [45] showed that it is possible to break the privacy offered by HTTP/2 multiplexing. HTTP/2 allows concurrent server threads to process multiple objects, resulting in multiplexed object transmission. This feature is useful for avoiding Head-of-Line blocking, i.e., a large object in the queue blocking the subsequent objects from being processed, which has been widely exploited in HTTP/1.1 to perform traffic analysis. Furthermore, HTTP/2 multiplexing makes it difficult for a passive attacker to identify individual objects over TLS traffic, and for this reason it is used as a basis for relevant privacy schemes. The authors assumed that an attacker may alter network parameters, namely latency, jitter, bandwidth, and packet drops, to introduce spacing between consecutive GET requests, which are sent to a server. This process can block the opportunity for the server to multiplex the objects corresponding to these requests, thus, negating the privacy benefits that come with it. The experimental results using the above parameters showed that:

- a uniform delay introduced for all packets is not effective for the described attack,
- the introduction of jitter so that the inter-arrival time of requests is 50ms results in 54% of objects not being multiplexed,
- a bandwidth reduction of 20% resulted in over 60% of non-multiplexed cases, and
- an 80% packet drop starting when the object of interest is sent for at least 6 sec resulted in 90% non-multiplexed cases.

As a remediation, the authors propose that some features of HTTP/2, namely server push and prioritization, can be used to set a different object priority and confuse the attacker. Regarding HTTP/3, it should be further examined if such attacks are still applicable, since the latest HTTP protocol version uses multiplexing and streams to transfer data.

2.7 Attack taxonomy

A taxonomy of the attacks reported in this section is presented in Figure 1. The attacks can generally be classified into two broad categories based on their HTTP version relevance: the ones that apply only in HTTP/2 and the ones that apply in HTTP/2 but could also apply in HTTP/3. Based on their characteristics, the studied attacks are classified into five categories: amplification, cryptojacking, DoS, slow-rate DoS, smuggling and privacy. As already explained above, even though slow-rate is a special subcategory of DoS, we chose to examine it separately due to being more difficult to detect.

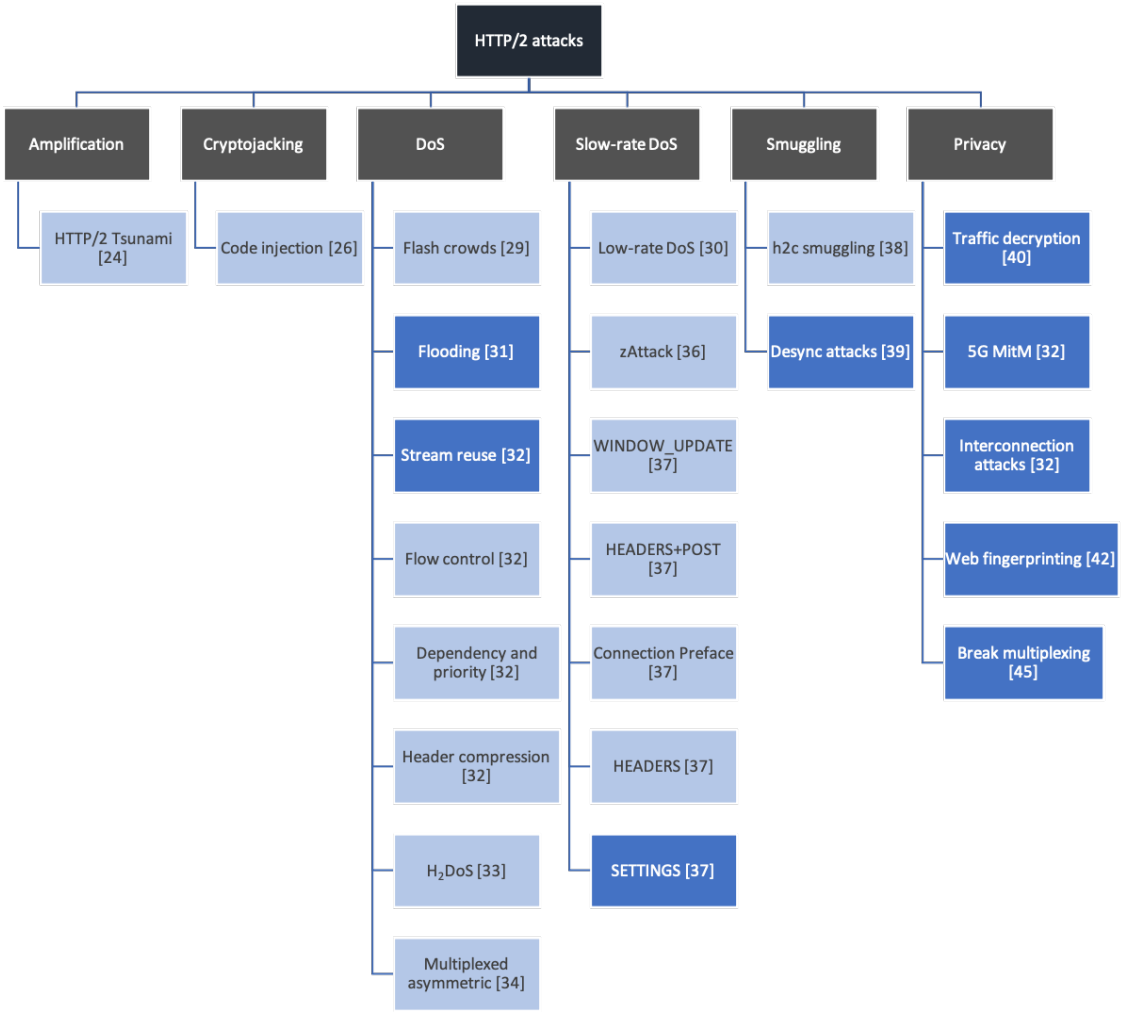

Figure 1: Taxonomy of attacks against HTTP (light blue: attacks against HTTP/2 only, blue: attacks against HTTP/2 that apply also to HTTP/3)

A first observation is that the majority of attacks (around 63%) are DoS (DoS and slow-rate DoS in Figure 1). Another major remark is that about one third of the attacks (close to 38%) can be ported to HTTP/3, mainly due to the different mechanisms used for flow control. Regarding individual categories, all the privacy-related attacks can be ported to HTTP/3, showing that the different characteristics of the new protocol version do not affect privacy-intrusive methods.

3 Hands on evaluation

For the hands-on evaluation, we reused the contemporary testbed given in § 5.1 of [46]. Precisely, this testbed is composed of the currently six most popular QUIC- and HTTP/3-enabled server implementations, namely OpenLiteSpeed, Caddy, NGINX, H2O, IIS, and Cloudflare. The reader should keep in mind that, at the time of writing, paradoxically, some servers like Algernon [47] do endorse QUIC, without however supporting HTTP/3. In total, four attacks were tested against each server; two of them stem directly from the HTTP/2 literature, that is, flooding and slow-rate, while the rest, that is, downgrade and HTTP/3-tables/streams that are presented in subsections 3.2 and 3.4, are new. The results per attack on each server implementation are recapitulated in Table 1. The relevant attack scripts are available at a public GitHub repository[2].

[2] `https://github.com/efchatz/HTTP3-attacks`

Table 1: Vulnerabilities identified in each server

Name	OpenLiteSpeed	Caddy	NGINX	H2O	IIS	Cloudflare	Total
HTTP/3 flooding	✗	✓	✗	✗	✗	✓	2
HTTP/x downgrade	✓	✓	✓	✓	✓	✓	6
Slow-rate HTTP/3 POST	✗	✓	✗	✗	✗	✗	1
HTTP/3-tables/streams	✓/✗	✓/✓	✓/✓	✓/✓	✓/✗	✗/✓	6
Total issues	2	5	3	3	2	3	–

3.1 HTTP/3 flooding attack

For this attack, the *curl* library with HTTP/3 enabled was used; note that at the time of writing, HTTP/3 and QUIC support in curl is unripe. To this end, we relied on the Docker image provided in *curl-http3*[3] repository in GitHub. First, we built locally the docker image by using the Dockerfile of that repository. Next, through the `docker exec` command, we connected to the Docker and executed the attack.

The attack used the *bash* capabilities to utilize 10 parallel curl requests; each request was executed for 1 sec (timeout). The curl command had the *GET* method as the primary one along with three additional method headers, namely, *HEAD, POST*, and *GET*. Also, a custom "settings" header with the 0 value was included, together with 26 bytes of null data to be sent along with each HTTP request.

On Caddy, the result of this tactic is a CPU usage of 99.9% 30 sec after initiating the attack, thus, paralyzing the server. Moreover, after the attack was active for 30 sec, Cloudflare presented an increased delayed response time of more than 3 sec with each request. The remaining servers coped well with this attack, i.e., they only suffered a heightened ($<$5%) CPU usage.

3.2 HTTP/x downgrade attack

Each server has been specifically setup to only communicate with the HTTP/3 protocol. However, it was observed that if TCP data were allowed to pass the firewall, the server responded to HTTP/1.1 requests establishing an HTTP/1.1 connection. Interestingly, the server admin is provided with no option to disable HTTP/1.1, but only to block the TCP protocol via the firewall. In this respect, if the firewall allows TCP traffic, the attacker may be able to mount HTTP/1.1 protocol relevant attacks, such as, HTTP request smuggling ones [21, 39].

Even worse, in addition to HTTP/1.1 connections, three out of the six servers, namely H2O, IIS 10, and Caddy, allowed HTTP/2 connections (without being configured as such), thus further increasing the server's attack surface. It can be argued that, for the sake of backwards compatibility, enabling by default HTTP/x protocols is desirable. However, this capability should be offered to the server admin in an opt-in/opt-out basis, which is not the case for the affected servers.

3.3 Slow-rate HTTP/3 POST attack

Another slow-rate type of attack was tested against all the servers, this time using a different HTTP method, namely the *POST* one. The attack script initiates about 40 parallel connections, with each one terminated after 5 sec. For this attack, we also employed the *OpenSSL* library for generating custom and random payloads of 32 bytes, which were sent to the targeted server. Caddy was the only server affected by this attack variation; the server's CPU usage was increased, thus delaying its responses to the clients trying to fetch a webpage.

3.4 HTTP-tables/streams attack

The current attack tampers with the values of *max_table_capacity* and *blocked_streams* parameters. These two parameters were introduced with the new control HTTP/3 frames, and they are transmitted with the so-called *SETTINGS* frame as the last fields related to HTTP/2. Note that the default values for these two parameters are 4096 bytes and 16 streams, respectively. We experimented with both small and large values for these fields, namely, 16 bytes and 4 streams and 409,600 bytes and 1,600 streams, respectively. By exploiting the *aioquic* Python library, we created 100 parallel connections to the targeted server with a timeout of 5 sec. This means that some connections were dropped before their completion.

Each attack lasted for about 2 min. Regarding the parameters' small values, IIS 10 and H2O presented a significant delay of around 3 sec in their HTTP responses. On the other hand, OpenLiteSpeed and Nginx paralyzed, being unresponsive

[3] `https://github.com/unasuke/curl-http3`

for about 10 to 30 sec. Cloudflare seems to be largely immune to this attack, nevertheless, all the servers but Caddy suffered an increased CPU usage between 10% and 15% while the attack was ongoing. Even worse, the CPU usage in the Caddy server reached 99.9% just after the first sec of the attack.

Similar observations were made when testing higher values for these two fields. Cloudflare presented a delay that exceeded 3 sec, but only for new connections. H2O suffered an additional response time of more than 4 sec, Caddy showed a CPU usage of 99.9%, responding with an excessive delay to new and existing connections, and Nginx was paralyzed, being unresponsive for about 10 to 30 sec. It can be assumed that these issues are related to HTTP/3 (or even QUIC) libraries used by each server, and they are a clear indication that new implementations need further examination before their deployment in real-life environments.

Given the severity of this attack, following a Coordinated Vulnerability Disclosure (CVD) process, we informed the affected vendors about the underlying vulnerability. To track this issue, MITRE assigned CVE-2022-30592, which received a base Score of 9.8 (critical)[4]. At the time of writing, only LiteSpeed has released a patch in *lsquic*[5] to mitigate this issue, which basically triggers a Null Pointer Dereference. Precisely, the fix comes in the form of zeroing the value of any *max_table_capacity* parameter that is lower than 32.

4 Dataset

As already pointed out, in the context of this work, and in view of the results presented in Section 3 and in § 5.2 of [46], we create the first to our knowledge dataset considering attacks on HTTP/2, HTTP/3, and QUIC. A preliminary evaluation of the dataset by means of legacy Machine Learning methods is also offered. We anticipate that the publicly provided dataset[6] along with its evaluation will serve as a common basis and guidance for future work.

The dataset, dubbed "H23Q" comprises a total of 10 assaults:

- Those given in Table 1, but the HTTP/x downgrade one.
- The *quic-flooding*, *quic-encapsulation*, *quic-loris*, and *quic-fuzz* assaults introduced in § 5.2 of [46].
- An HTTP request smuggling attack plus two traditional attacks from the HTTP/2 domain: (i) A flooding one, which sets a hefty *max_concurrent_request* value equal to 100K, and (ii) a pause-resume flooding, which repeatedly creates HTTP/2 connections; the connections are paused and then resumed. The latter two attacks were included for the sake of completeness, since no work so far offer an HTTP/2 security-oriented dataset. It should be noted that the HTTP-request smuggling assault has a similar effect to the HTTP/x downgrade one.

The H23Q dataset is offered in both pcap and CSV formats. Precisely, the CSV files are labelled and contain 200 features, i.e., 199 generic ones and the label class. We also include several Python scripts and instructions on how to generate new CSV files with additional set of features and how to label them.

4.1 Testbed

The testbed for the creation of the dataset comprised six different HTTP/3-enabled servers, which run on the Azure cloud infrastructure. The hardware specifications of all the employed machines, servers and clients, are summarized in Table 2. The utilized clients were operated from three different subnetworks. The first comprised six clients, the second three, and the last four, with one of them operated by the attacker. Each client and server received its last update on April 30, 2022.

To replicate real-life traffic scenarios, two of the public network interfaces (DSL routers), changed their public (routable) IP address during the recording process. This means that some attacks contain different public IP addresses for the same clients. Regarding the configuration of each deployed server, the interested reader is referred to § 5.1 of [46]. Note that IIS 10, H2O, and Caddy enable HTTP/2 by default.

Figure 2 depicts a high-level view of the network topology. The red-colored client in the third subnetwork represents the attacker, while the orange-colored ones are part of the botnet the attacker created for the needs of specific attacks.

Each server was behind a DNS zone. The latter was assigned with a registered domain name, and then, each server was assigned with a unique subdomain. Each HTTP server offered a simple HTML webpage. For some indefinite reason,

[4] `https://nvd.nist.gov/vuln/detail/CVE-2022-30592`

[5] `https://github.com/litespeedtech/lsquic/releases/tag/v3.1.0`

[6] The "H23Q" dataset is available for download at the well-known AWID website at `https://icsdweb.aegean.gr/awid/other-datasets/H23Q`.

Table 2: Specifications of servers and clients in the testbed

Name	OS	CPU/RAM	Version	Network	IP address (router)
OpenLiteSpeed	Ubuntu 18.04	1/1	1.7.15	Azure	10.0.0.4
Caddy	Ubuntu 18.04	1/1	2.4.6		10.0.0.5
NGINX	Ubuntu 18.04	2/4	1.21.7		10.2.0.4
H2O	Ubuntu 18.04	1/1	2.3.0-DEV		10.0.0.4
IIS	Windows Server 2022	2/4	10		10.0.0.5
Cloudflare	Ubuntu 18.04	2/4	1.16.1		10.1.0.4
Client 1	Windows 11	2/4	21H2	1	5.203.250.215/ 5.203.228.219/ 5.203.253.63
Client 2	Windows 11	2/4	21H2		
Client 3	Windows 10	2/4	21H2		
Client 4	Windows 10	2/4	21H2		
Client 5	Ubuntu 22.04	2/4	5.15.0-27		
Client 6	Ubuntu 22.04	2/4	5.15.0-27		
Client 7	Windows 11	2/4	21H2	2	5.203.165.59/ 5.203.215.191/ 5.203.255.68
Client 8	Windows 10	2/4	21H2		
Client 9	Windows 10	2/4	21H2		
Client 10	Windows 10	2/4	21H2	3	85.75.109.194
Client 11	Windows 10	2/4	21H2		
Client 12	Ubuntu 22.04	2/4	5.15.0-27		
Attacker	Ubuntu 20.04	4/16	5.13.0-40		

some servers, including IIS 10, experienced problems in offering every time its webpage over HTTP/3. So, these servers communicated with the clients with either HTTP/2, if they supported it by default, or HTTP/1.1. As a result, in the dataset, apart from the HTTP/3 normal traffic, one can observe HTTP/2 and HTTP/1.1 normal traffic as well.

Client-server communication was done based on a random pattern. Precisely, the *Selenium* Python library was installed in each client, and each one of them requested randomly to connect to an HTTP server. Then, the client waited for 5 sec, and retried to communicate with another or the same server after a random sleep time ranging between 1 and 5 sec. The HTTP connection was made either through the Chrome or the Firefox browser. To enable the decryption of the recorded traffic, all the clients, including the attacker's one, stored their TLS keys locally.

4.2 Data collection

Regarding the data collection, the following points are important:

- The network traffic capturing process was performed on each server separately. The reason behind this choice is that by recording traffic in this way, it offers more flexibility in terms of a possible IDS, either a network-based or host-based IDS. To this end, each attack was split into six different pcap files, one per server. And since the dataset contains 10 attacks, the dataset comprises 60 pcap files.
- To reduce the size of the dataset, we recorded around 1M packets per attack, meaning that each server captured approximately 150K packets.
- As mentioned in Section 4.1, each client stored their TLS keys. This enables the decryption of the corresponding traffic in the dataset.
- The *Wireshark* v3.6.3 utility was installed on each server for capturing the incoming and outgoing traffic. For Ubuntu-based servers, *tshark* was used; the latter utilizes Wireshark to capture the traffic. All the captured processes were filtered by appropriate IP and port filters for not recording any unwanted traffic.
- The attack parameters, including its duration, the frames per second rate, and the use of bots or not, differ depending on the attack type. The purpose was to trace out each attack the best way possible.

4.3 Attacks in the dataset

As already mentioned, the dataset comprises 60 pcap files, that is, 10 attacks $\times$ in 6 servers. Each attack against a server was performed following the same order, i.e., OpenLiteSpeed, Caddy, NGINX, IIS, Cloudflare, and H2O. We

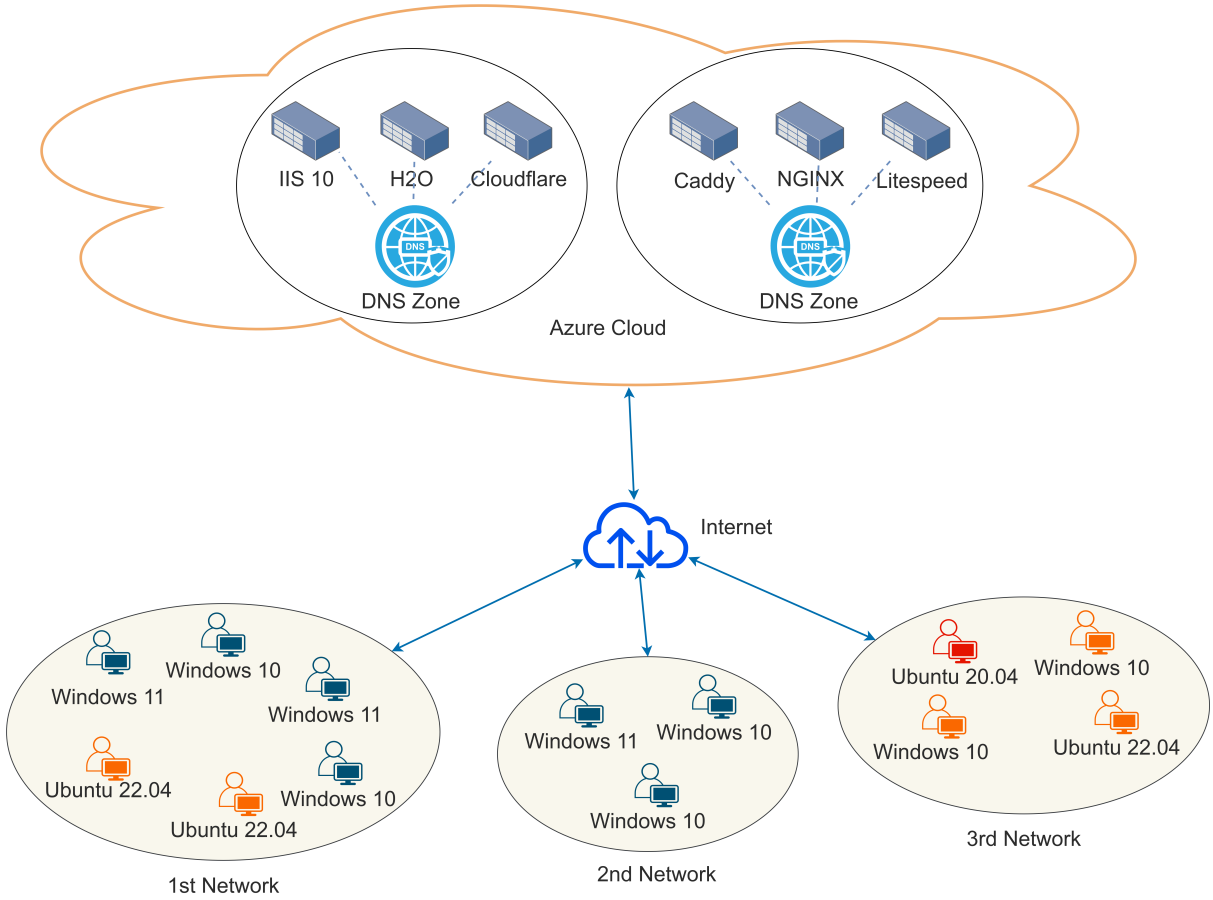

Figure 2: Network topology used in the creation of the dataset. The red node refers to the attacker, while the orange ones to the devices that belong to the attacker's botnet.

detail each attack below, while Table 3 recaps the characteristics of each attack as seen in the corresponding files of the dataset.

- *HTTP-flood*: Multiple HTTP/3 requests were sent to each server. To achieve this, the attacker utilized *curl* v7.83.0 along with the local bots for creating a DDoS effect. The total time of this assault was 10 min, with the first 4 being normal traffic. After that, each server was under attack for 1 min.
- *Fuzzing*: It contains fuzzing traffic and a number of packets similar to those of a *hash-collision* attack [48]. The opponent attacked solo, without the use of bots. Again, the assault lasted for 10 min, with the first 4 being normal traffic and the remaining 6 devoted to attacking each server for 1 min. The attacker utilized the *Fuzzotron* fuzzer along with the *Scapy* Python library for crafting custom packets. This assault should be considered as a transport layer one, since most of the attack traffic consists of UDP datagrams.
- *HTTP-loris*: This assault has the same duration as the previous two. The basic difference here is that the attacker utilized both local and remote bots, thus, causing a significant DDoS effect. The local bots issued simple HTTP requests, while the remote bots and the attacker were placing an HTTP request with a random payload every 5 sec; the data were generated through the *OpenSSL* library.
- *HTTP-stream*: This attack was done in two cycles, carried out back to back. The first, follows the same timing scheme as the previous three assaults: the first 4 min for normal traffic and the rest 6 for the attack. In this cycle, the attacker utilized the *aioquic* Python library v0.9.20, and as mentioned in Section 3.4, they increased by far the *max_table_capacity* and *blocked_streams* values. The second cycle comprises 3 min of normal traffic, and after that, the attacker mounted a variation of the assault where the aforementioned two fields had a low value. The total duration of this attack is 20 min, with the attacks laying from the 4th to the 10th, and from

the 13th to the 19th min. Note that this assault is directly related to CVE-2022-30592. So, it can be possibly used as a zero-day attack, because OpenLiteSpeed was unpatched for this issue during the attack's execution.

- *QUIC-flood*: With the aid of the *aioquic* library, the attacker instructed the local bots to perform a QUIC-flood. The timing scheme of the current attack is identical to the first three ones.
- *QUIC-loris*: The attacker exploited both the botnets, therefore, increasing the DDoS effect. The connection requests crafted with the help of the *aioquic* library were placed every 5 sec from the attacker and both the local and remote bots. The attack phase is between the 4th and the 10th minutes. Note that this assault is directly related to CVE-2022-30591. So, it can be possibly used as a zero-day attack, because Caddy was unpatched for this issue during the attack's execution.
- *QUIC-enc:* The methodology of the current assault is similar to the *quic-encapsulation* one detailed in [46]. Through the *Scapy* library, the attacker sends custom encapsulated packets, which are in the form of *IP(UDP(IP(TCP)))* and *IP(UDP(IP(UDP)))*. The timing scheme of the current attack is identical to the first three ones.
- *HTTP-smuggling*: It has a longer duration, namely, 15 min. The aggressor initiated the attack at the 3rd min and assaulted persistently each server for 2 min. For this attack, the attacker utilized *curl* along with *OpenSSL* for generating custom payloads per packet. A different packet structure was used when attacking each server.
- *HTTP/2-concurrent*: This penultimate attack pertains to HTTP/2. Its total duration was 6 min, with the attacker launching it after the 3rd min, and changing the targeted server every 30 sec. Once more, the *curl* tool was used. Specifically, for stressing each server, the tool was instructed to create 100K *MAX_TOTAL_CONNECTIONS* with 100K *MAX_CONCURRENT_STREAMS* each; recall that the first variable defines the maximum number of simultaneous open connections of a client, while the second, the maximum count of simultaneous streams to support over a single HTTP connection. In case the target server did not enable HTTP/2, the traffic was over HTTP/1.1, thus resulting to an HTTP/1.1 flooding.
- *HTTP/2-pause*: This last assault has the same timing scheme as the previous one. Through the *curl* tool, the assailant ceases and starts the HTTP/2 threads of each connection in an attempt to paralyze the server. If the target server did not offer HTTP/2, the attack takes the form of an HTTP/1.1 flooding.

4.4 Signature of attacks

This section offers symptomatic signatures (footprints) of selected attacks of the dataset based on packet per second (PPS) units of measurement. Specifically, we chose four representative assaults, i.e., two HTTP/3 and two QUIC oriented. The former were taken from a specific server, while the latter depict the traffic from all the servers.

First off, Figure 3 depicts the normal versus HTTP-flooding traffic, but only for the Cloudflare server. As it can be observed, the attack is clearly differentiated against the normal traffic. Second, Figure 4 footprints an HTTP-loris assault exercised against the OpenLiteSpeed server. Such "under the radar" assaults are typically used with the purpose of bypassing certain network perimeter protection mechanisms. Indeed, as observed from the figure, the attack pattern is almost identical to that of the normal traffic. So, identifying such an attack, especially on a single server, is quite challenging.

Third, Figure 5 illustrates the QUIC-flood assault done against the six servers. A higher number of packets were captured between the 400 and 500 sec, possibly targeting the IIS server. As with the HTTP-flood one, the attack pattern is quite easily distinguishable if compared to that of the normal traffic. Last but not least, the QUIC-loris assault is illustrated in Figure 6. The footprint of this attack seems to be clearer in comparison to that of the HTTP-loris, possibly due to the combination of traffic stemming from all the six servers.

5 Dataset evaluation

In this section, we perform an initial evaluation of the H23Q dataset through machine learning techniques, both shallow and deep learning. First, we detail the feature selection and data preprocessing procedures, and then elaborate on the experiments and the derived results. The experiments were performed on an MS Windows 10 Pro machine with AMD Ryzen 7 2700 CPU and 64 GB RAM. For shallow classifiers, we only rely on the CPU; no GPU was utilized. We employed the *sklearn* v.1.0.1 in Python v3.8.10, for all classifiers and metrics, except from *LightGBM*. The latter algorithm was implemented with the homonymous Python library in v.3.3.2.

5.1 Feature selection and data preprocessing.

The following points are important regarding feature selection and data preprocessing.

Table 3: Details per attack per server. A separate pcap and CSV file is given per attack per server. An asterisk means that this attack produced a DDoS effect. The 3rd column designates the analogy of malicious to normal traffic, whereas the 7th column the malicious to total traffic. The two "traffic" columns and the "Normal/Malicious" one are expressed in number of packets.

Per attack			Per server					
Attack name	Normal/Malicious	%	Server	Total traffic	Malicious traffic	%	Size (MB)	Duration
HTTP-flood*	1,316,770/498,810	37.88	LiteSpeed	329,264	112,466	34.15	281	6/10
			Caddy	142,218	64,082	45.05	125	
			NGINX	175,211	71,588	40.85	164	
			IIS	146,020	87,720	60.07	131	
			Cloudflare	342,777	76,747	22.38	319	
			H2O	181,278	86,207	47.55	195	
Fuzzing	660,412/22,224	3.36	LiteSpeed	191,456	2,440	1.27	157	6/10
			Caddy	47,703	3,085	6.46	55	
			NGINX	79,849	3,175	3.97	76	
			IIS	55,666	6,799	12.21	61	
			Cloudflare	227,949	2,552	1.11	225	
			H2O	57,787	4,173	7.22	75	
HTTP-loris*	677,240/74,572	11.01	LiteSpeed	217,093	24,060	11.08	180	6/10
			Caddy	77,846	14,579	18.72	79	
			NGINX	79,436	7,210	9.07	73	
			IIS	56,631	9,566	16.89	59	
			Cloudflare	184,311	15,713	8.52	177	
			H2O	61,922	3,444	5.56	75	
HTTP-stream	1,318,226/1,063	0.08	LiteSpeed	310,957	436	0.14	286	12/20
			Caddy	125,561	52	0.04	144	
			NGINX	186,542	27	0.01	176	
			IIS	141,770	453	0.31	153	
			Cloudflare	381,959	47	0.01	374	
			H2O	171,435	48	0.02	213	
QUIC-flood*	746,608/61,340	8.21	LiteSpeed	199,894	8,215	4.10	184	6/10
			Caddy	72,366	7,798	10.77	80	
			NGINX	105,826	8,345	7.88	101	
			IIS	66,448	23,210	34.92	71	
			Cloudflare	206,436	6,550	3.17	202	
			H2O	95,637	7,222	7.55	120	
QUIC-loris*	653,451/24,269	3.71	LiteSpeed	225,238	15,075	6.69	193	6/10
			Caddy	55,237	51	0.09	61	
			NGINX	73,417	759	1.03	71	
			IIS	45,158	1,699	3.76	54	
			Cloudflare	188,099	6,632	3.52	186	
			H2O	66,301	53	0.07	85	
QUIC-enc	741,467/5,829	0.78	LiteSpeed	224,651	689	0.30	191	6/10
			Caddy	66,115	847	1.28	76	
			NGINX	108,556	1,300	1.19	104	
			IIS	49,456	1,091	2.20	58	
			Cloudflare	221,603	1,006	0.45	217	
			H2O	71,085	896	1.26	90	
HTTP-smuggle	1,331,394/3,337	0.25	LiteSpeed	252,837	749	0.29	247	12/15
			Caddy	122,690	536	0.43	139	
			NGINX	180,606	342	0.18	173	
			IIS	83,507	540	0.64	90	
			Cloudflare	381,765	403	0.10	378	
			H2O	129,382	425	0.32	168	
HTTP2-concurrent	982,768/60,273	6.13	LiteSpeed	188,046	3,997	2.12	134	3/6
			Caddy	211,210	30,694	14.53	173	
			NGINX	225,822	3,437	1.52	209	
			IIS	127,480	21,567	16.91	73	
			Cloudflare	184,388	175	0.09	181	
			H2O	45,821	403	0.87	63	
HTTP2-pause	1,141,326/53,549	4.69	LiteSpeed	339,280	3,043	0.89	342	3/6
			Caddy	235,461	23,827	10.11	191	
			NGINX	238,000	2,880	1.21	225	
			IIS	138,203	23,466	16.97	148	
			Cloudflare	142,630	227	0.15	143	
			H2O	47,751	109	0.22	63	

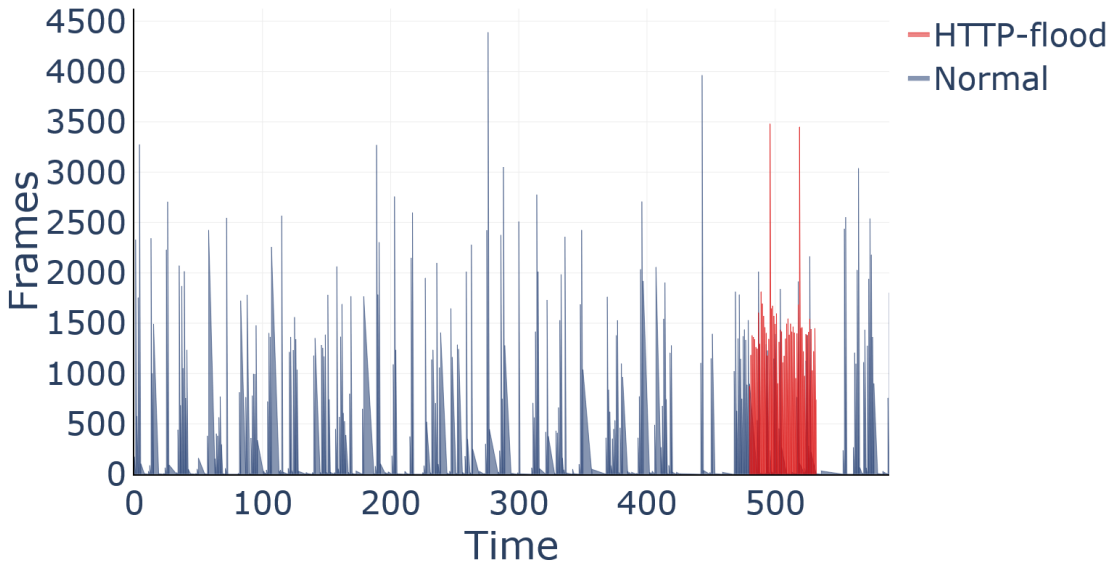

Figure 3: HTTP-flood footprint on the Cloudflare server

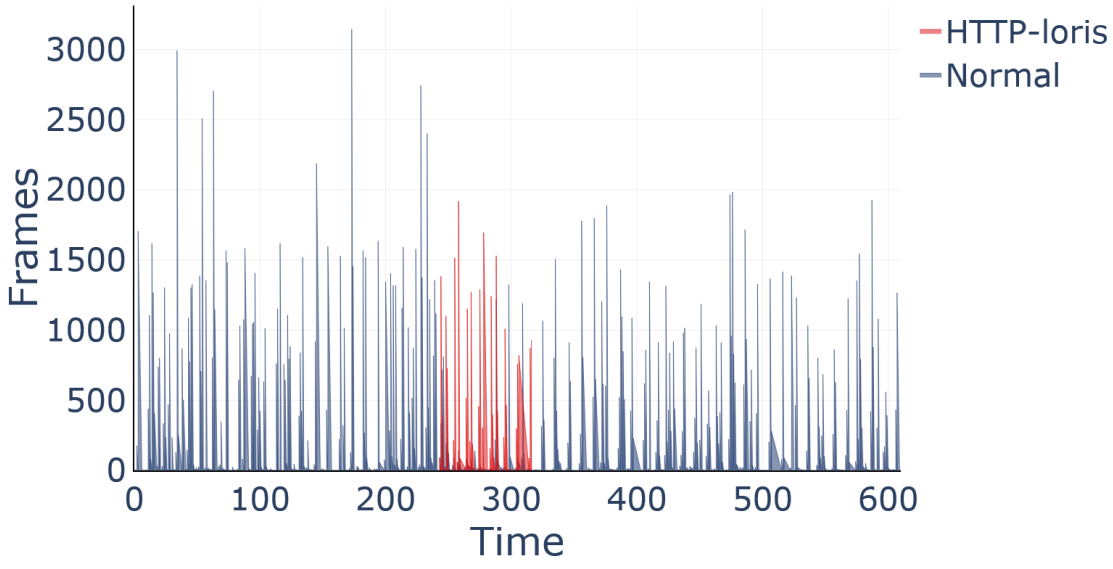

Figure 4: HTTP-loris footprint on the OpenLiteSpeed server

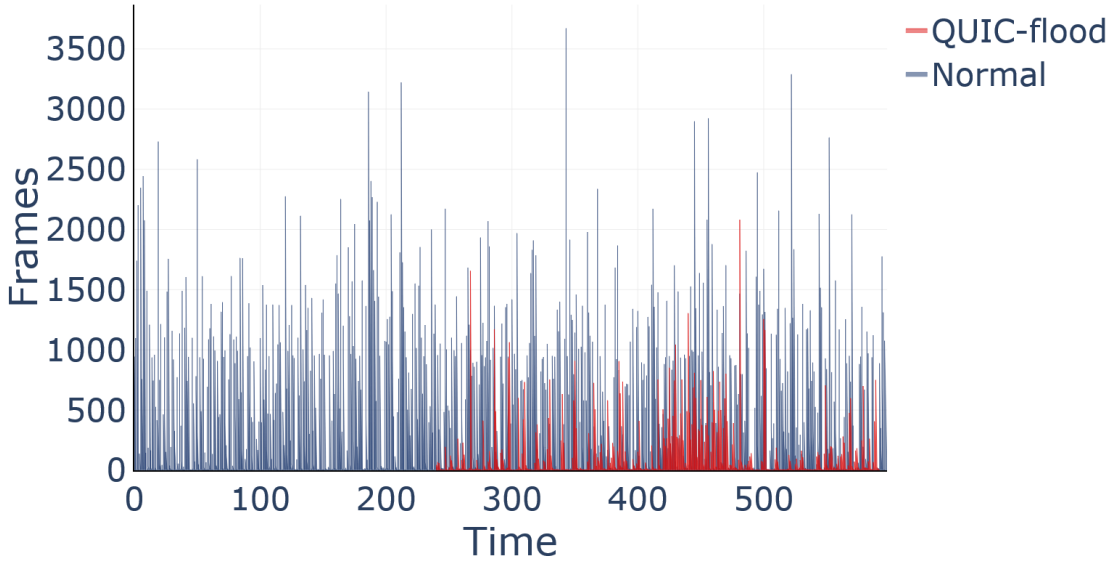

Figure 5: QUIC-flood footprint on all servers

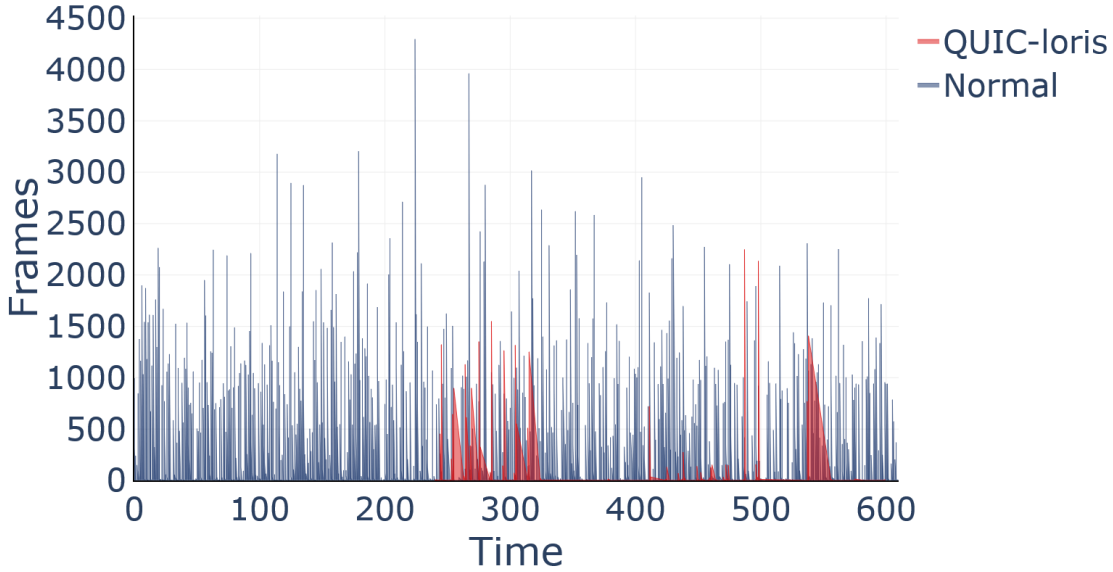

Figure 6: QUIC-loris signature on all servers

- First, a large number of features (199) were extracted from the pcap files. This was a provisional set of features that could possibly assist in the identification of malicious traffic. To extract these features, we utilized *tshark*. However, before running *tshark*, the TLS keys of each client were loaded from *Wireshark*. Then, via *tshark*, we extracted the decrypted traffic of each pcap file, finally getting the initial set of the 199 features. From this large set of features, we cherry-picked less than half of them, i.e., a total of 46 features. The feature selection process was based on the study of previous work summarized in § 3.1 of [49].
- In a next step, the labelling process added one more feature to designate the attack class. Note that the Azure Cloud anonymizes the MAC addresses of the incoming traffic, therefore, every MAC address in each pcap file is anonymized as "12:34:56:78:9a:bc"; this applies only to clients, not servers. There was no option to disable this protection, so for the dataset to reflect an even more realistic traffic.
- The 46 features were split into two categories, namely, *Constant* and *Discrete*. That is, if a feature had constant values, it was scaled with the Min-Max technique. On the other hand, if the feature had discrete values, the One-Hot-Encoding (OHE) technique was used. Most of the features converted with the Min-Max scheme presented values with the scientific notation; these features were rounded to three decimal points. Some features, like the *quic.length* one, carried multiple values in the same frame. The respective values of these features were added into a single value.
- Due to the existence of multiple diverse protocols, namely TCP, UDP, QUIC, HTTP, and others, picking features from different protocols could result in having multiple empty cells. For this reason, we tinkered with the *Constant* features by replacing empty cells with 0, and with the *Discrete* ones by replacing empty cells with -1. This was done to clearly differentiate between the two states, i.e., *Constant*/empty vs. *Discrete*/empty.
- Finally yet importantly, we divided the 10 attacks into five classes, namely, *Normal, DDoS-flooding, DDoS-loris, Transport-layer attacks*, and *HTTP/2 attacks*, having the homonymous labels. The *DDoS flooding* class comprises the HTTP-flood, HTTP-stream, and the QUIC-flood attacks. The *DDoS-loris* class consists of the HTTP-loris and the QUIC-loris assaults. The *Transport-layer* class includes the Fuzzing and QUIC-enc, while the *HTTP/2 attacks* class contains the HTTP-smuggle, HTTP/2-concurrent, and HTTP/2-pause assaults.

  For easy reference, the finally selected 46 features along with the utilized data preprocessing method per feature are given in Table 4. The left column designates the feature name as it was exported from *tshark*.

Table 4: List of features used to evaluate the dataset

Feature name	**Preprocessing**
frame.len, ip.len, tcp.len, tcp.hdr_len, tcp.window_size_value, tcp.option_len, udp.length, tls.record.length, tls.reassembled.length, tls.handshake.length, tls.handshake.certificates_length, tls.handshake.certificate_length, tls.handshake.session_id_length, tls.handshake.cipher_suites_length, tls.handshake.extensions_length, tls.handshake.client_cert_vrfy.sig_len, quic.packet_length, quic.packet_number_length, quic.length, quic.nci.connection_id.lengt, quic.crypto.length, quic.stream.len, quic.token_length, quic.padding_length, http2.length, http2.header.length, http2.header.name.length, http2.header.value.length, http2.headers.content_length, http3.frame_length, http3.settings.qpack.max_table_capacity, http3.settings.max_field_section_size, dns.count.queries, dns.count.answers, http.content_length	Min-Max
tcp.flags.ack, tcp.flags.push, tcp.flags.reset, tcp.flags.syn, tcp.flags.fin, quic.long.packet_type, quic.fixed_bit, quic.spin_bit, quic.stream.fin, dns.flags.response, http.content_type	OHE
Label	–

5.2 Experiments

For this initial evaluation of the dataset, we relied on commonly accepted ML techniques, without resorting to any optimization or dimensionality reduction schemes. Given that the dataset is imbalanced, the focus was on the AUC and F1 scores. Bear in mind that in the experiments, the 49-feature set of Table 4 was used.

5.2.1 Shallow classification analysis

A number of common classifiers in the IDS domain were considered. For determining the optimal parameters, the *GridSearchCV* algorithm was used. *GridSearchCV* divides a dataset into x-fold partitions and evaluates the placed

parameters for finding out the optimal ones. A 2-fold validation scheme was employed for evaluating the F1 score. The best results were obtained with the LightGBM, Decision Trees (DT), and Bagging classifiers. We did not use cross validation; the dataset was analyzed in a 60/40% fashion for the training and test sets, respectively. For equally splitting the dataset, the stratified split scheme was used.

Table 5: Parameter values per Shallow classifier. A hyphen denotes that the current value is impertinent to the current ML algorithm.

Parameters	**DT**	**LightGBM**	**Bagging**
max_depth	200	1000	–
max_leaf_nodes	1000	–	–
min_samples_leaf	2	–	–
min_samples_split	10	–	–
ccp_alpha	0.0001	–	–
max_bin	–	2000	–
min_child_samples	–	30	–
min_data_in_bin	–	50	–
min_split_gain	–	0.1	–
n_estimators	–	1000	500
num_leaves	–	3000	–
learning_rate	–	0.01	–
reg_alpha	–	0.001	–
reg_lambda	–	0.001	–
n_jobs	–	1	–
max_samples	–	–	100,000

Table 6 presents the results per utilized classifier in terms of the AUC, Precision, Recall, F1-Score, and Accuracy (Acc) scores. The total time of each model's execution in hours/min/sec is also included in the rightmost column of the table. The Acc column is included just for the sake of completeness, and therefore is shown in gray font. The top score regarding the AUC and F1 metrics per classifier is highlighted with green, whereas the lowest with orange. As observed from the table, the best performer was the Bagging model with an AUC and F1 score of 77.60% and 68.77%, respectively. LightGBM performance was also very close to the Bagging one.

Figure 7 depicts the confusion matrix of the top performer. The classifier missed ≈37K packets of the *Normal* class, misclassifying them as *DDoS-flooding*. Similarly, ≈105K packets of the *DDoS-flooding* class were wrongly identified as *Normal*. This indicates that the best performer experienced difficulties in differentiating between these two classes. An equivalent situation was perceived for the three remaining classes, namely, *Transport-layer*, *DDoS-loris*, and *HTTP/2 attacks*, where the algorithm missed ≈2.5K, ≈35K, and ≈7K packets, respectively, misplacing them to the *Normal* class. Overall, these results indicate that the best performer missed the samples of every attack-class in a percentage ranging between ≈25 to 75%, misplacing the corresponding samples to the *Normal* class.

Table 6: Results on shallow classifiers

Model name	AUC	Prec.	Recall	F1	Acc	T.t.
DT	76.67	65.93	63.56	64.21	94.44	00:04:30
LightGBM	77.49	79.71	64.72	68.40	94.76	05:42:05
Bagging	77.60	79.95	64.81	68.77	94.82	03:07:43

5.2.2 DNN classification analysis

Three different models, namely, Multi-layer Perceptron (MLP), Denoising stacked Autoencoders (AE), and TextCNN were employed for DNN analysis. The parameters used per DNN model are recapped in Table 7. To obtain full control over the training phase, the mini-batch Stochastic Gradient Descent (SGD) optimizer was implemented, with a learning rate of 0.01 and a momentum of 0.9. Having said that, a low *Batch* size, e.g., 150 can result in a more generalized DNN model; this is because more data will be analyzed during each *Epoch*. In this respect, a *Batch* size of 256 was used. Moreover, where applicable, the well-known *ReLU* activator was utilized. Another common activator function for the output layer of DNN is the so-called *Softmax*. The latter was implemented to classify the results. No less important, a regularization effect was added through the *Dropout* scheme.

The input layer for TextCNN was diverse, namely the number of the dataset rows, while the output designated the number of classes, that is, five. Moreover, the same DNN model employed the Embedding layer of *Keras*, and has been

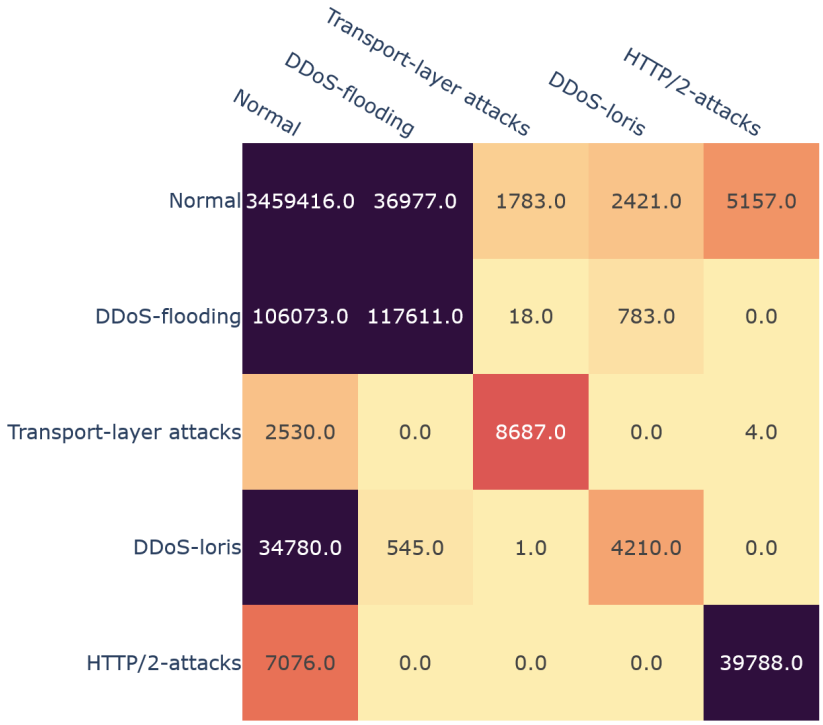

Figure 7: Confusion matrix for the Bagging algorithm

utilized with three *Conv1D* hidden layers using the same padding. The *AveragePooling1D* layer was implemented after the first two hidden layers, while the *GlobalAveragePooling1D* was added after the third hidden layer. For both models, the *BatchNormalization* layer was applied after each hidden layer.

Additional techniques, including *Model Checkpoint* and *Early Stopping*, were applied to preserve the optimal training state of each DNN model. For these two schemes, we checked for the minimum loss value, and if the DNN model did not improve their loss value for two consecutive epochs, the training phase was ceased and the model was re-trained with the last optimal epoch. This eventually means that every fold was trained for a minimum two more epochs. These options, alongside other techniques, including *Dropout* and *validation test*, conceivably retained overfitting to the bare minimum.

The results in terms of the AUC metric per examined model are presented in Table 8. The penultimate column of the table indicates the number of epochs needed by the relevant DNN model to be trained. As observed from the table, in terms of the AUC metric, the AE and MLP models presented the best and worst performance scores, respectively (about 68.3% vs. 58.7%). Overall, the current scores lag behind vis-à-vis the scores yielded by shallow classification; nearly -9.30% and -15% for the AUC and F1 metrics, respectively. On the other hand, the TextCNN model was by far the fastest one for both set of features, requiring ≈5 hours. To provide a clearer picture of the results, Figure 8 depicts the accuracy and validation performance of loss per epoch.

The above-mentioned inferior outcome is corroborated by the numbers in Figure 9, which presents the confusion matrix of the top performer. It is easily perceived that, similar to the results of shallow classification, MLP experienced the same or even worse issues regarding the classification of the samples belonging to all the attack classes. Precisely, a higher percentage of samples of the *DDoS-flooding*, *DDoS-loris*, and *HTTP/2-attacks* classes – around 50%, 89%, and 15%, respectively – have been misplaced as *Normal* traffic.

Table 7: Parameter values per DNN algorithm. A value of "/3" or "/4" in the MLP Dropout parameter indicates the number of layers in which this parameter received the designated value. The layer values are calculated without including the input and output ones. SCC stands for Sparse Categorical Crossentropy. A hyphen denotes a non-applicable option for this DNN model.

Parameters	**MLP**	**Autoencoders**	**TextCNN**
Activator	ReLU	ReLU	ReLU
Output activator	Softmax	Softmax	Softmax
Initializer	He_uniform	–	–
Optimizer	SGD	SGD	SGD
Momentum	0.9	0.9	0.9
Dropout	0.25/3-0.2/4	0.25	0.2
Learning rate	0.01	0.01	0.01
Loss	SCC	SCC	SCC
Batch Norm.	✓	✓	✓
Embedding layer	✗	✗	✓
Hidden layers	7	12	4
Nodes (Per layer)	300/200/160/120 /60/30/10	300/200/160/120/60/30/ 10/30/60/120/160/200/300	Conv1D(64,12)/ Conv1D(128,10)/ Conv1D(256,12) Dense(200)
Batch size	256	256	256

Table 8: Results of DNN model analysis

Model name	AUC	Prec.	Recall	F1	Acc	Epochs	T.t.
MLP	68.34	62.15	51.15	53.71	92.62	68	07:16:34
AE	58.72	63.50	33.15	39.59	92.51	48	08:19:29
TextCNN	64.52	72.44	45.9	46.71	92.37	42	05:12:10

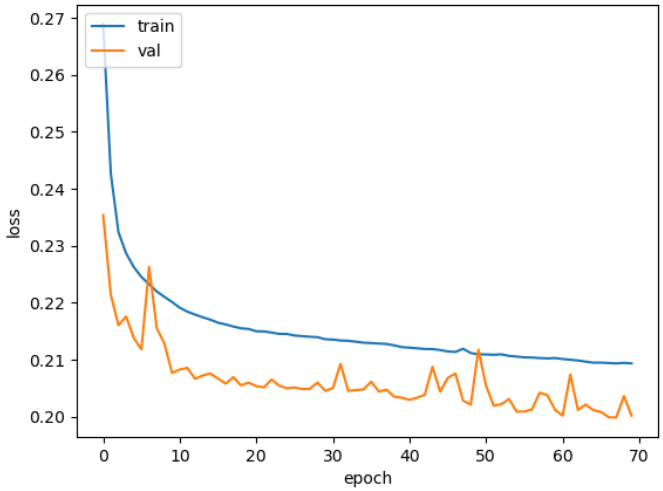

Figure 8: MLP model comparison of training loss with the validation loss, after each epoch

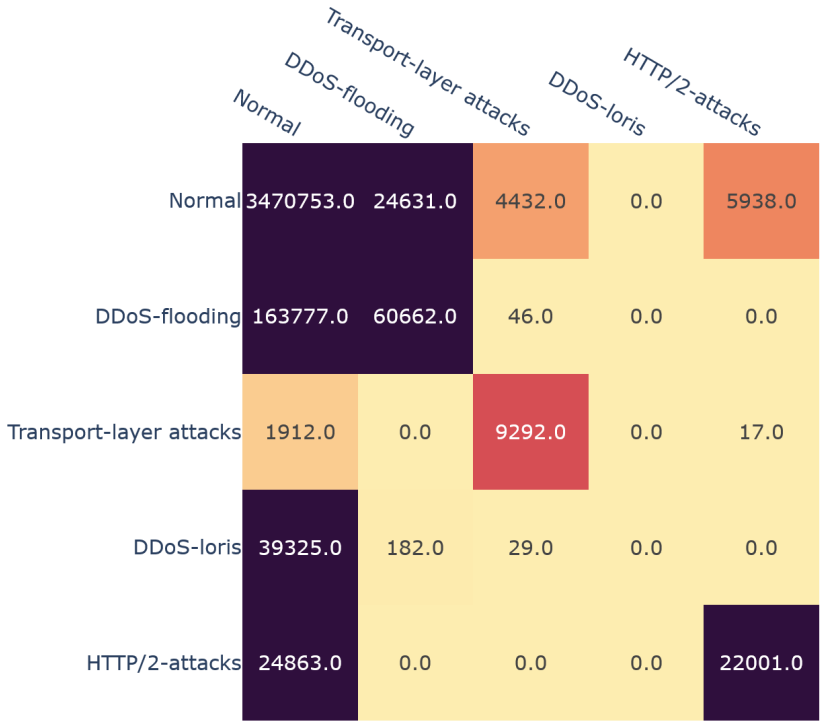

Figure 9: MLP confusion matrix

5.2.3 Anomaly detection

Finally yet importantly, we analyzed the AE model through the anomaly detection method. While this model presented the worst results in Section 5.2.2, we chose it because, according to the literature, it is the commonest approach to anomaly detection. To this end, the dataset samples were divided into two classes, namely, *Normal* and *Malicious*. In a next step, using the stratified split scheme, the dataset was split into three subsets, i.e., *Train*, *Test*, and *Validation*, each having the 50%, 30%, and 20% of the dataset samples, respectively. The *Train* subset contained samples from only the *Normal* class, while both the *Test* and *Validation* subsets comprised samples from both classes. Therefore, the training phase was performed solely over samples of the *Normal* class.

The *Label* feature was removed from the *Test* subset, and kept to a separate subset, i.e., a *Label Test* subset. This was done to validate the results with the reconstruction error. The AE model was configured as in Table 7; the only difference was that the output layer contained one node because it had the *Linear* function as the output of the AE model. The loss function (SCC) was also replaced with the *Mean Absolute Error* (MAE). To make sure that the training phase did not suffer from overfitting, we compared the training loss against the validation loss values after each epoch. Indeed, as shown in Figure 10, the training phase did not exhibit overfitting. The model was trained for 13 epochs, and produced a MAE of 0.3228 after the last epoch.

After the training phase, the model was evaluated by calculating the reconstruction error. For this purpose, first, the model was requested to predict the *Test* subset. Then, the MAE was computed by taking the absolute difference between the derived predictions and the *Test* subset values. To identify which of these predictions was correctly chosen, i.e., to compare and separate the MAE error, the *Label* subset was used. Precisely, if the label for a sample was *Malicious*, the corresponding MAE error was flagged as an anomaly, while the remaining samples were flagged as *Normal*. By trial and error, it was calculated that the threshold for this analysis should be 0.33. Figure 11 depicts this observation, highlighting that the model was mostly unable to discern between the two classes. This means that a better feature selection process is probably needed, as already mentioned in Section 5.2.2.

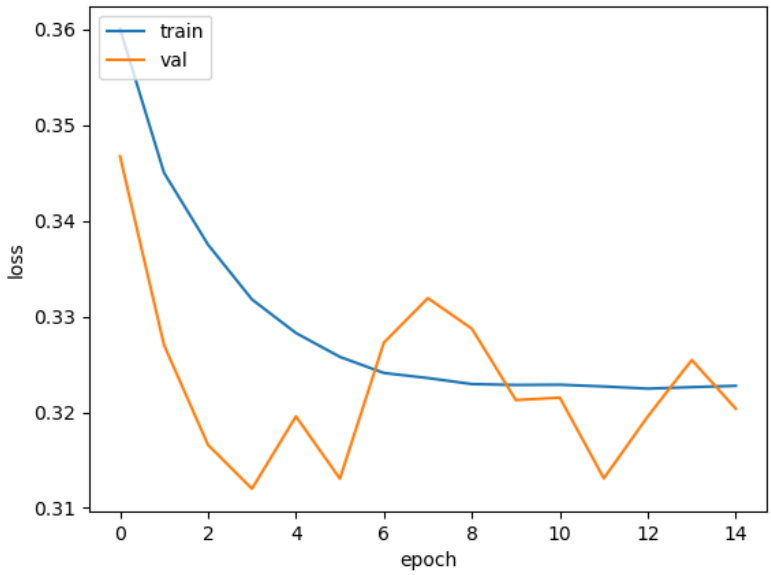

Figure 10: Anomaly detection model comparison of training loss with the validation loss, after each epoch

Moreover, Figure 12 illustrates the prediction rate of the current model. For generating this confusion matrix, we labeled every sample above the threshold as *Malicious*, while the rest of the samples were marked as *Normal*. As seen from the figure, the results are imprecise, i.e., the *Malicious* class is largely confused with the *Normal*. On the other hand, the latter class misplaced a rather small percentage of the samples (≈3.4%) as *Malicious*. Regarding legacy evaluation metrics, namely, Precision, Recall, F1-score, and Acc, they presented tolerable results, i.e., 91.75%, 96.60%, 94.11%, and 88.94%, respectively. Obviously, this behavior, i.e., a lower Acc score vis-à-vis F1, is due to the imbalanced nature of the dataset.

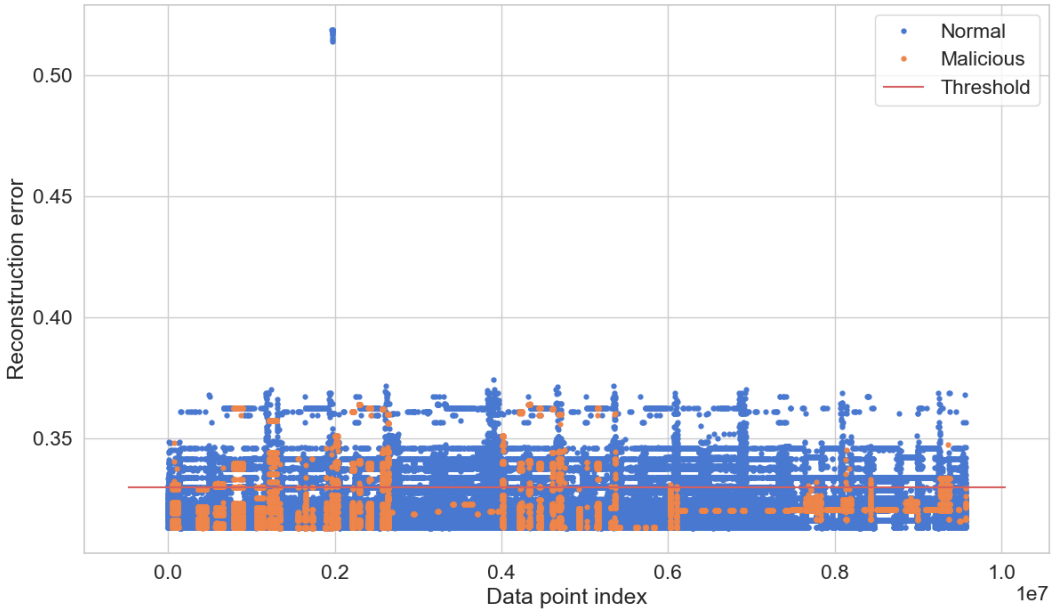

Figure 11: Anomaly detection model – reconstruction error with threshold

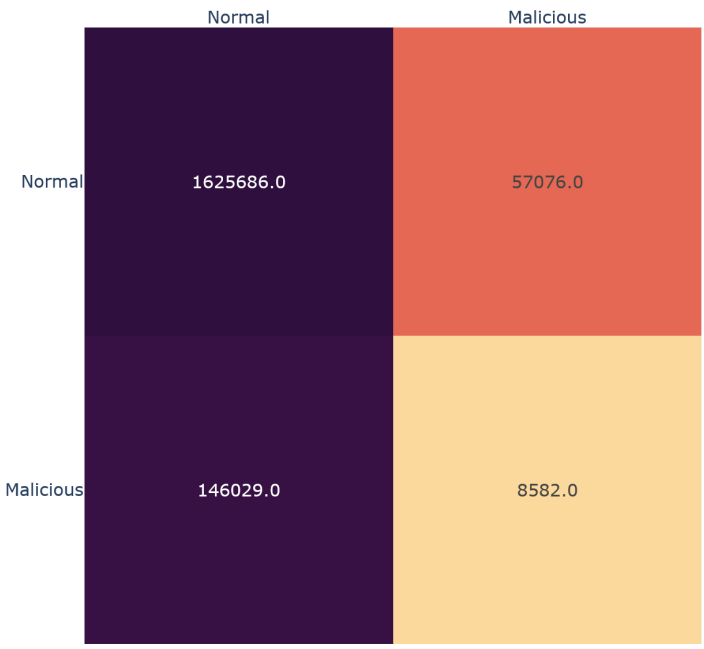

Figure 12: Anomaly detection model confusion matrix based on threshold

6 Conclusion

The work at hand delivers the first to our knowledge full-fledged study on HTTP/2 security, extending the identified attacks to its successor, namely HTTP/3. Specifically, starting with a review of HTTP/2 attack categories, we examine half a dozen of contemporary HTTP/3-enabled servers regarding their resilience against either common or uncommon attack tactics. This endeavor yielded interesting results, some of which leading to CVE. What is more, through the creation of a realistic testbed, we created a rich, voluminous (30 GB) dataset containing an assortment of 10 attacks against HTTP/2, HTTP/3, and QUIC. The dataset, coined "H23Q", is labeled and is offered publicly to the community. A preliminary evaluation of the dataset conducted by means of different techniques on a set of 46 cross-layer features revealed, as expected, that certain attack classes are very challenging to detect. In this respect, future work can concentrate on both cherry-picking of more informative features and the use of more sophisticated IDS techniques, including network flow analysis and time series anomaly detection.

Other possible avenues for future work include: (i) the analysis of HTTP/2 Websockets [50] from a security perspective; note that the bootstrapping of WebSockets with HTTP/3 is just around the corner [51], and (ii) the development of a full-featured HTTP/x fuzzer, enabling meticulous vulnerability testing. Thus far, the only HTTP/2-focused fuzz tool is the so-called *http2fuz* [52], which however is quite outdated.

References

[1] Martin Thomson and Cory Benfield. *HTTP/2*. RFC 9113. June 2022. DOI: 10.17487/RFC9113. URL: https://www.rfc-editor.org/info/rfc9113.

[2] Efstratios Chatzoglou, Georgios Kambourakis, and Christos Smiliotopoulos. "Let the Cat out of the Bag: Popular Android IoT Apps under Security Scrutiny". In: *Sensors* 22.2 (2022), p. 513. DOI: 10.3390/s22020513. URL: https://doi.org/10.3390/s22020513.

[3] Efstratios Chatzoglou, Georgios Kambourakis, and Vasileios Kouliaridis. "A Multi-Tier Security Analysis of Official Car Management Apps for Android". In: *Future Internet* 13.3 (2021), p. 58. DOI: 10.3390/fi13030058. URL: https://doi.org/10.3390/fi13030058.

[4] Vasileios Kouliaridis et al. "Dissecting contact tracing apps in the Android platform". In: *Plos one* 16.5 (2021), e0251867. DOI: 10.1371/journal.pone.0251867.

[5] Georgios Karopoulos et al. "A Survey on Digital Certificates Approaches for the COVID-19 Pandemic". In: *IEEE Access* 9 (2021), pp. 138003–138025. DOI: 10.1109/ACCESS.2021.3117781.

[6] Mike Bishop. *HTTP/3*. RFC 9114. June 2022. DOI: 10.17487/RFC9114. URL: https://www.rfc-editor.org/info/rfc9114.

[7] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000.

[8] Georgios Karopoulos, Dimitris Geneiatakis, and Georgios Kambourakis. "Neither Good nor Bad: A Large-Scale Empirical Analysis of HTTP Security Response Headers". In: *Trust, Privacy and Security in Digital Business - 18th International Conference, TrustBus 2021, Virtual Event, September 27-30, 2021, Proceedings*. Vol. 12927. Lecture Notes in Computer Science. Springer, 2021, pp. 83–95. DOI: 10.1007/978-3-030-86586-3_6. URL: https://doi.org/10.1007/978-3-030-86586-3%5C_6.

[9] Franco Tommasi, Christian Catalano, and Ivan Taurino. "Browser-in-the-Middle (BitM) attack". In: *Int. J. Inf. Secur.* 21 (2022), pp. 179–189. DOI: 10.1007/s10207-021-00548-5. URL: https://doi.org/10.1007/s10207-021-00548-5.

[10] mr.d0x. *Steal Credentials & Bypass 2FA Using noVNC*. visited on 2022-03-22. URL: https://mrd0x.com/bypass-2fa-using-novnc/.

[11] mr.d0x. *Browser In The Browser (BITB) Attack*. visited on 2022-03-22. URL: https://mrd0x.com/browser-in-the-browser-phishing-attack/.

[12] W3Techs. *Historical trends in the usage statistics of site elements for websites*. last visited 22/06/2022. URL: https://w3techs.com/technologies/history_overview/site_element/all.

[13] Georgios Kambourakis and Georgios Karopoulos. "Encrypted DNS: The good, the bad and the moot". In: *Computer Fraud & Security* 2022.5 (2022), null. URL: https://doi.org/10.12968/S1361-3723(22)70572-6.

[14] Google. *DNS-over-HTTP/3 in Android*. last visited 22/07/2022. URL: https://security.googleblog.com/2022/07/dns-over-http3-in-android.html?m=0.

[15] Orange Tsai. "A New Era of SSRF-Exploiting URL Parser in Trending Programming Languages". In: *Black Hat* (2017).

[16] Snyk Security Research team. *URL confusion vulnerabilities in the wild: Exploring parser inconsistencies*. visited on 2022-03-22. URL: https://snyk.io/blog/url-confusion-vulnerabilities/.

[17] Noam Moshe, Sharon Brizinov, and Team82. *Exploiting URL Parsing Confusion*. visited on 2022-03-22. URL: https://claroty.com/2022/01/10/blog-research-exploiting-url-parsing-confusion/.

[18] Omer Gil. "Web cache deception attack". In: *Black Hat USA* 2017 (2017).

[19] Seyed Ali Mirheidari et al. "Cached and Confused: Web Cache Deception in the Wild". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 665–682. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/mirheidari.

[20] "Web Cache Deception Escalates!" In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/mirheidari.

[21] Efstratios Chatzoglou, Georgios Kambourakis, and Constantinos Kolias. "Your WAP Is at Risk: A Vulnerability Analysis on Wireless Access Point Web-Based Management Interfaces". In: *Secur. Commun. Networks* 2022 (2022), 1833062:1–1833062:24. DOI: 10.1155/2022/1833062. URL: https://doi.org/10.1155/2022/1833062.

[22] Parth Patni et al. "Man-in-the-middle attack in HTTP/2". In: *2017 International Conference on Intelligent Computing and Control (I2C2)*. 2017, pp. 1–6. DOI: 10.1109/I2C2.2017.8321787.

[23] Mingming Zhang et al. "Talking with Familiar Strangers: An Empirical Study on HTTPS Context Confusion Attacks". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1939–1952. ISBN: 9781450370899. URL: https://doi.org/10.1145/3372297.3417252.

[24] David Beckett and Sakir Sezer. "HTTP/2 Tsunami: Investigating HTTP/2 proxy amplification DDoS attacks". In: *2017 Seventh International Conference on Emerging Security Technologies (EST)*. 2017, pp. 128–133. DOI: 10.1109/EST.2017.8090411.

[25] Internet Engineering Task Force (IETF). *RFC7541 | HPACK: Header Compression for HTTP/2*. visited on 2022-04-06. URL: https://datatracker.ietf.org/doc/html/rfc7541.

[26] Meenakshi Suresh et al. "Exploitation of HTTP/2 Proxies for Cryptojacking". In: *Security in Computing and Communications*. Ed. by Sabu M. Thampi et al. Singapore: Springer Singapore, 2020, pp. 298–308.

[27] braiins.com. *Mining protocol | Stratum v2*. visited on 2022-04-06. URL: https://braiins.com/stratum-v2.

[28] Internet Engineering Task Force (IETF). *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. visited on 2022-04-06. URL: https://datatracker.ietf.org/doc/html/rfc7230.

[29] Erwin Adi et al. "Distributed denial-of-service attacks against HTTP/2 services". In: *Clust. Comput.* 19.1 (2016), pp. 79–86. DOI: 10.1007/s10586-015-0528-7.

[30] Erwin Adi et al. "Low-Rate Denial-of-Service Attacks against HTTP/2 Services". In: *2015 5th International Conference on IT Convergence and Security (ICITCS)*. 2015, pp. 1–5. DOI: 10.1109/ICITCS.2015.7292994.

[31] David Beckett and Sakir Sezer. "HTTP/2 Cannon: Experimental analysis on HTTP/1 and HTTP/2 request flood DDoS attacks". In: *2017 Seventh International Conference on Emerging Security Technologies (EST)*. 2017, pp. 108–113. DOI: 10.1109/EST.2017.8090408.

[32] Xinxin Hu et al. "Signalling Security Analysis: Is HTTP/2 Secure in 5G Core Network?" In: *2018 10th International Conference on Wireless Communications and Signal Processing (WCSP)*. 2018, pp. 1–6. DOI: 10.1109/WCSP.2018.8555612.

[33] Xiang Ling et al. "H$$_{2}$$DoS: An Application-Layer DoS Attack Towards HTTP/2 Protocol". In: *Security and Privacy in Communication Networks*. Ed. by Xiaodong Lin et al. Cham: Springer International Publishing, 2018, pp. 550–570. ISBN: 978-3-319-78813-5.

[34] Amit Praseed and P. Santhi Thilagam. "Multiplexed Asymmetric Attacks: Next-Generation DDoS on HTTP/2 Servers". In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 1790–1800. DOI: 10.1109/TIFS.2019.2950121.

[35] Tanishka Shorey et al. "Performance Comparison and Analysis of Slowloris, GoldenEye and Xerxes DDoS Attack Tools". In: *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 2018, pp. 318–322. DOI: 10.1109/ICACCI.2018.8554590.

[36] Yihang Zhang and Yijie Shi. "A Slow Rate Denial-of-Service Attack Against HTTP/2". In: *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*. 2018, pp. 1388–1391. DOI: 10.1109/CompComm.2018.8780763.

[37] Nikhil Tripathi and Neminath Hubballi. "Slow rate denial of service attacks against HTTP/2 and detection". In: *Computers & Security* 72 (2018), pp. 255–272. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2017.09.009.

[38] Jake Miller (bishopfox). *h2c Smuggling: Request Smuggling Via HTTP/2 Cleartext (h2c)*. visited on 2022-04-07. URL: https://bishopfox.com/blog/h2c-smuggling-request.

[39] James Kettle. *HTTP/2: The Sequel is Always Worse*. visited on 2022-03-14. URL: https://portswigger.net/research/http2.

[40] Meenakshi Suresh et al. "An investigation on HTTP/2 security". In: *Journal of Cyber Security and Mobility* 7 (Jan. 2018), pp. 161–180. DOI: 10.13052/jcsm2245-1439.7112.

[41] Vyron Kampourakis et al. "Revisiting man-in-the-middle attacks against HTTPS". In: *Network Security* 2022.3 (2022), null. DOI: 10.12968/S1353-4858(22)70028-1. eprint: https://doi.org/10.12968/S1353-4858(22)70028-1. URL: https://doi.org/10.12968/S1353-4858(22)70028-1.

[42] Weiran Lin, Sanjeev Reddy, and Nikita Borisov. "Measuring the Impact of HTTP/2 and Server Push on Web Fingerprinting". In: *Workshop on Measurements, Attacks, and Defenses for the Web*. 2019.

[43] Amazon. *The top 500 sites on the web*. visited on 2022-03-22. URL: https://www.alexa.com/topsites.

[44] Jean-Pierre Smith, Prateek Mittal, and Adrian Perrig. "Website Fingerprinting in the Age of QUIC". In: *Proceedings on Privacy Enhancing Technologies* 2021.2 (2021), pp. 48–69. DOI: doi:10.2478/popets-2021-0017.

[45] Gargi Mitra et al. "Depending on HTTP/2 for Privacy? Good Luck!" In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2020, pp. 278–285. DOI: 10.1109/DSN48063.2020.00044.

[46] Efstratios Chatzoglou et al. "Revisiting QUIC attacks: A comprehensive review on QUIC security and a hands-on study". In: *Res. Sq. Prepr.* (2022), pp. 1–22. DOI: 10.21203/rs.3.rs-1676730/v1.

[47] Algernon. *Small self-contained pure-Go web server with Lua, Markdown, HTTP/2, QUIC, Redis and PostgreSQL support*. visited on 2022-06-18. URL: https://github.com/xyproto/algernon.

[48] Bart Preneel. "Collision Attack". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 220–221. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_564. URL: https://doi.org/10.1007/978-1-4419-5906-5_564.

[49] Efstratios Chatzoglou et al. "Best of Both Worlds: Detecting Application Layer Attacks through 802.11 and Non-802.11 Features". In: *Sensors* 22.15 (2022). ISSN: 1424-8220. DOI: 10.3390/s22155633. URL: https://www.mdpi.com/1424-8220/22/15/5633.

[50] Patrick McManus. *Bootstrapping WebSockets with HTTP/2*. RFC 8441. Sept. 2018. DOI: 10.17487/RFC8441. URL: https://www.rfc-editor.org/info/rfc8441.

[51] Ryan Hamilton. *Bootstrapping WebSockets with HTTP/3*. Internet-Draft draft-ietf-httpbis-h3-websockets-04. Work in Progress. Internet Engineering Task Force, Feb. 2022. 5 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-h3-websockets-04.

[52] Stuart Larsen and John Villamil. "Attacking HTTP/2 Implementations". In: *13th PACific SECurity-Applied Security Conferences and Training in Pacific Asia (PacSec)*. 2015.