PRETT2: Discovering HTTP/2 DoS Vulnerabilities via Protocol Reverse Engineering

Choongin Lee[1], Isa Jafarov[2], Sven Dietrich ✉[2], and Heejo Lee ✉[1]

[1] Korea University, Seoul, Republic of Korea
{choonginlee, heejo}@korea.ac.kr
[2] City University of New York, New York, USA
ijafarov@gradcenter.cuny.edu, spock@ieee.org

Abstract. HTTP/2, enhancing data transmission speed over HTTP/1.1 with features such as flow control for stream multiplexing, has seen widespread adoption across major web servers. This has exposed numerous vulnerabilities, with denial of service (DoS) particularly prominent due to flawed flow control implementations. Identifying potential weaknesses in the flow control across various HTTP/2 implementations has largely depended on manual inspection. However, the behavioral diversity among implementations poses significant challenges for testing.
In this study, we propose PRETT2, a stateful fuzzing framework targeting denial-of-service (DoS) vulnerabilities in HTTP/2 protocols. Utilizing automated protocol reverse engineering, PRETT2 infers state machines unique to various HTTP/2 implementations. Then it executes multiplexed fuzzing that manipulates flow control messages based on the identified state machines. Testing on servers such as Apache and Nginx revealed the capability of PRETT2 to infer multiple state machine types and uncover security vulnerabilities, including CVE-2023-43622 by Apache. This highlights the effectiveness of PRETT2 in identifying and addressing critical security vulnerabilities in HTTP/2.

1 Introduction

HTTP/2 [45] is an advanced application-level protocol that supersedes HTTP/1.1 and includes additional features. According to the Cloudflare global web traffic statistics [9], HTTP/2 traffic accounts for the majority of the HTTP protocol series (around 65%). The primary advantage of HTTP/2 is that it supports flow control for stream multiplexing [57, 58]. This addresses the limitations of HTTP/1.1, such as Head-of-Line blocking [40], and enhances the protocol's efficiency and performance.

Unfortunately, the number of vulnerability reports regarding HTTP/2 implementations has steadily increased as HTTP/2 software is developed with various security holes that may be exploited. Figure 1 illustrates the number of Common Vulnerabilities and Exposures (CVEs) of HTTP/2 implementations since the introduction of HTTP/2 in 2015. Among these vulnerabilities, 74% are classified as

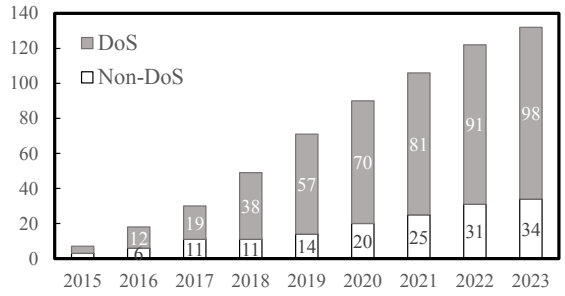


Fig. 1: The accumulated number of CVEs for HTTP/2 software per year. The majority type of CVEs is DoS. The CVEs are selected based on their descriptions that contain the keyword `HTTP/2`.

denial-of-service (DoS) attacks [30] in 2023[1]. Multiple studies have criticized that HTTP/2 implementations are vulnerable to DoS attacks [1, 8, 22, 24, 29, 48, 50] due to the inappropriate use of features for flow control. We highlight the point that the incomplete implementation of flow control features could potentially compromise network security.

To mitigate potential HTTP/2 DoS attacks, it is imperative to conduct a thorough analysis of HTTP/2 implementations to identify potential threats that may exhaust system resources via flow control. To do this, security analysts are required to have a deep understanding of complex communication features. Although prior endeavors using manual analysis methods have discovered many DoS vulnerabilities, they are daunting and time-consuming to apply for diverse implementations. Even worse, an HTTP/2 implementation typically consists of a significant amount of source code which makes it hard to analyze by hand. As a remedy, automated testing can significantly reduce the effort.

For the automated testing of HTTP/2, a few methods [7, 41] have been proposed. Nevertheless, it is still challenging to discover DoS vulnerabilities in HTTP/2 due to the sophisticated properties of HTTP/2. Based on our empirical analysis, we set three key challenges for automated testing: flow control, stateful testing, and scalability.

1. ***Flow control:*** Flow control is a vital feature of HTTP/2, enabling stream multiplexing. We examined that most CVE-registered vulnerabilities are attributed to security flaws in the flow control. Thus, it is important to take into account its multiplexing features to identify deficiencies.
2. ***Stateful testing:*** Stateful fuzzing is a method that enables stateful testing, which tests server under test (SUT) with test cases based on state information [35]. To analyze potential security flaws of protocol implementation, it is effective to use stateful testing that uses a finer-grained state machine [7].

[1] We determined the type of a CVE as DoS if its description contains related keywords such as denial-of-service, application/server crashes, or excessive resource consumption as its effect.

Unlike HTTP/1.1, HTTP/2 is a stateful protocol, in which the acceptable set of messages and the observable response depend on each protocol state.

3. ***Scalability:*** We examined that around 100 implementations of HTTP/2 are available on the GitHub source code repository [23] and numerous versions per implementation. We examined that HTTP/2 traces for the same user behavior can vary across servers and clients, including their versions, indicating significant variations in the protocol's state machines. The diversity can be attributed to the preferences or intentions of developers [17, 21]; they may exclude or add certain features. Hence, stateful fuzzing for HTTP/2 must be sufficiently scalable to accommodate the diverse range of state machines.

This paper introduces PRETT2, a stateful fuzzing framework designed to analyze HTTP/2 DoS via multiplexed fuzzing. Addressing the aforementioned challenges, PRETT2 infers its state machines by the use of protocol reverse engineering given network traces. Interacting with the server, PRETT2 utilizes existing work [28] for the protocol reverse engineering with a modified fuzzing algorithm that supports messages of binary protocol and checks time feedback. Based on the inferred state machines, PRETT2 traverses state transitions and performs multiplexed fuzzing, generating test cases with manipulated flow control messages. During fuzzing, PRETT2 measures another time feedback and compares it to previously measured feedback to detect extended connection durations, assessing potential DoS vulnerabilities.

The experimental results show that PRETT2 successfully inferred multiple types of state machines depending on the implementations. Through the stateful fuzzing HTTP/2 implementations, such as Apache and Nginx, PRETT2 identified three security flaws resulting in DoS, which have been reported to the developers. One of the vulnerabilities has been assigned the CVE-ID (CVE-2023-43622) by Apache.

This work presents the following contributions:

- We present a stateful fuzzing framework to automatically analyze HTTP/2 denial-of-service flaws. It uses a novel method to infer HTTP/2 state machines and performs fuzzing tests targeting its flow control features.
- Our experiments show that diverse state machines can be inferred for the same user behavior depending on the implementations. Notably, we demonstrate that the flow control of a state machine may be different from others.
- We discovered and reported multiple security flaws in the flow control of popular HTTP/2 implementations using stateful fuzzing by PRETT2.

2 Background

2.1 HTTP/2

HTTP (Hypertext Transfer Protocol) is a fundamental protocol used to request data from web servers. The first version of HTTP for common usage was HTTP/1 [10], followed by HTTP/1.1 [20]. One of the significant improvements in

Table 1: Properties of HTTP/1.1 and HTTP/2

	HTTP/1.1	*HTTP/2*
Message format	plain text	binary frames
Connections	pipelining	stream multiplexing
Statefulness	stateless	stateful
Transport layer	TCP	TCP

HTTP/1.1 is its ability to handle parallelized requests through pipelining. However, HTTP/1.1 still had a limitation known as head-of-line blocking (HoLB) [40] which causes delays in simultaneous data transmission and cannot cope with huge network traffic. To be specific, a response to a request has to wait until the response to the previous request is processed when requests are in parallel. To address this issue, a new protocol called SPDY was developed, forming the basis for HTTP/2 [15, 43].

Table 1 outlines the distinctions between HTTP/2 and HTTP/1.1. A key difference is the shift of HTTP/2 from plain text to binary framing for HTTP headers and messages, optimizing communication data efficiency. Unlike HTTP/1.1's reliance on multiple TCP connections, HTTP/2 employs stream multiplexing within a single TCP connection to enable parallel requests and responses. For instance, HTTP/2 allows for setting stream priorities and maintaining idle streams with the `PRIORITY` frame [57, 58]. Additionally, HTTP/2 introduces statefulness with enhanced functionalities like flow control, in contrast HTTP/1.1 has a stateless nature [45]. Such property makes the sequences of messages with appropriate frames important. The list of all HTTP/2 frames and the detailed information is described in Table 2.

Table 2: Types of HTTP/2 communication frames

Frame type	Description
SETTINGS	Configures flow control parameters for the communication in a stream.
WINDOW_UPDATE	Sets or updates a stream's data capacity for flow control.
DATA	Transmits variable-length octet sequences in a stream.
HEADERS	Opens a stream for data request or response.
PUSH_PROMISE	Allows multiple responses to a single request from the client.
PRIORITY	Specify a stream to be processed first by designating the priority.
RST_STREAM	Immediately terminates a stream with an error code.
GOAWAY	Terminates the connection using a priority error code over other frames.
CONTINUATION	Continues a prior HEADERS frame's field block sequence.
PING	Assesses idle connection functionality by measuring round-trip time.

Flow Control. Flow control in HTTP/2 facilitates the multiplexing of communications within a single connection by preventing the interference of streams between a server and a client. It leverages the `SETTINGS` and `WINDOW_UPDATE` frames to manage resource allocation through `DATA` frames. This involves configuring the receiver's buffering capacity, known as the flow control window, and allowing each sender to maintain distinct flow control windows for each stream and the connection as a whole [6]. Adjustments to the flow control window, such as setting initial sizes via `SETTINGS` frame and modifying current sizes through

Listing 1: HTTP/2 specification: Denial-of-Service Considerations

```
... The SETTINGS frame can be abused to cause a peer to expend
    additional processing time. This might be done by pointlessly
    changing SETTINGS parameters, setting
    multiple undefined parameters, or changing the same setting
    multiple times in the same frame. WINDOW_UPDATE or PRIORITY frames
     can be abused to cause an unnecessary waste of resources. ...
```

`WINDOW_UPDATE` frame, enable communication parties to signal their readiness to receive data [1].

2.2 Protocol Reverse Engineering

Protocol reverse engineering discerns the structure and sequence of protocol messages, resulting in a state machine that specifies the anticipated interactions within each state, aiding analysts in protocol behavior analysis [32]. Its key advantage lies in enabling fuzzers to conduct stateful testing by generating test cases that explore the complex state dynamics of an implementation.

Protocol reverse engineering can be done by hand, but it requires huge labor. Unfortunately, protocol state machines are often not well-defined in typical protocol specifications, according to numerous works of literature [26, 33, 36, 37]. Specifically, many protocol specifications are written in prose [26, 36] or minimally documented [32]. Further, multiple state machines may be inferred for the same protocol depending on implementation. Such diversity requires security experts to spend extra effort. As a remedy, automated protocol reverse engineering is effective in coping with the diverse state machines. Prior work has demonstrated that protocols can successfully be reverse-engineered in an automated way [11, 55], which is also possible to infer multiple state machines per implementation [28].

3 Security Analysis for HTTP/2

3.1 Prior Works

Manual Analysis. Prior works on the security analysis of HTTP/2 relied heavily on manual analysis [8, 29, 48, 50]. However, one limitation of such manual analysis is scalability. This is because HTTP/2 has multiple implementations with different features based on the developer's interpretations [25]. For instance, one paragraph in the `Denial-of-Service Considerations` section of the HTTP/2 specification (shown in Listing 1) does not specify the exact parameters and values (or their ranges) that correspond to the *multiple* numbers. The obscure descriptions may lead to using subjective mitigations.

Automated Analysis. As a remedy to manual analysis, proposals for automated analysis have emerged as effective means for identifying HTTP/2 vulnerabilities. However, despite these advances, they have yet to surmount all the challenges previously mentioned. The detailed description of each approach is as follows:

Table 3: Challenges addressed by automated analysis

	Flow Control	*Stateful Testing*	*Scalability*
http2fuzz	Yes	No	No
SGFUZZ	No	Yes	Yes
Proposed	**Yes**	**Yes**	**Yes**

1. http2fuzz [41] is an open-source project that conducts fuzzing on HTTP/2 messages. It uses pre-defined message formats to generate test cases and supports HTTP/2 messages that include flow control frames such as `SETTINGS` and `WINDOW_UPDATE` frames. This addresses the challenge of *flow control*. However, due to a lack of insight into the state machines implemented in different implementations, it is unable to tackle the challenges of *stateful testing* and *scalability*.
2. SGFUZZ [7] is the first to use stateful fuzzing research for HTTP/2. It is effective at addressing challenge *stateful testing* by automatically extracting the state machine of target implementation (H2O). SGFUZZ uses static analysis on source code which is the general approach that can be applied to diverse implementations and addresses challenge *scalability*. However, the state machine extracted by SGFUZZ does not include HTTP/2 messages for flow control. It only contains state transitions for data requests and responses, which makes it unable to address challenge *flow control*.

The challenges addressed by each method and our proposal have been summarized in Table 3.

3.2 Motivating Example

As the main inspiration for our study, we describe an example of an HTTP/2 DoS vulnerability, which is publically reported as CVE-2016-1546 [16]. The vulnerability is triggered by an HTTP/2 slow rate attack [48] that exploits a flaw in the flow control implementation of the Apache HTTP/2 server. The sequence diagram demonstrating the HTTP/2 slow-rate attack is shown in Figure 2.

Slow-rate Attack. A slow-rate attack [12] is a subcategory of DoS attacks, which is mainly known for its stealthier nature. This attack is one of the well-known HTTP/2 vulnerabilities, which is pointed out by various researches [2,47–49,59]. Unlike large-scale DDoS attacks that can easily be noticed, a slow-rate attack

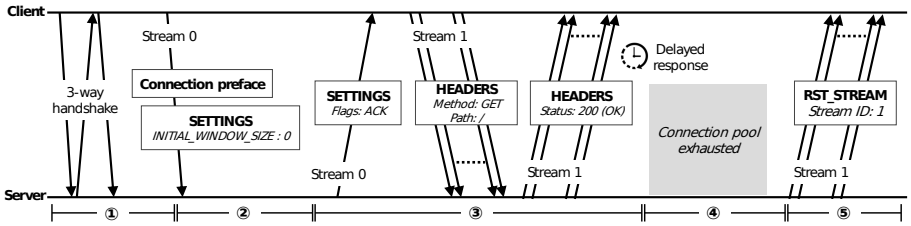


Fig. 2: Sequence diagram of HTTP/2 slow-rate attack

is hard to detect since it requires a small bandwidth and is similar to normal traffic. This attack proceeds according to the following steps:

① A TLS connection is established through a 3-way handshake, after which the client sends a connection preface on stream ID 0.
② Then it sends a `SETTINGS` frame that is deliberately crafted with an initial window size set to zero.
③ Following the receipt of another `SETTINGS` frame that includes the `ACK` flag, the attacker sends multiple `GET` requests using a `HEADERS` frame on a different stream, such as stream ID 1.
④ This leads to a communication deadlock, where the server is ready to send data but is artificially constrained due to the window size being set to zero, preventing any data transmission.
⑤ Consequently, the server keeps these connections open for an extended period, exhausting the connection pool until it terminates the stream with `RST_STREAM` frames, which could last longer than any configured timeout.

Proposed approach. Our analysis of HTTP/2 DoS vulnerabilities emphasizes the importance of addressing flow control, stateful interactions, and scalability. Assuming an attacker can interact with the target software in parallel, they can exploit complexities in *flow control* with *stateful* messages that can strain the server. Vulnerabilities may vary across different server types and versions, necessitating a testing approach that is both flexible and *scalable*. We adopt a black box approach that realistically simulates attacks using real-world network services. The limitations of white box analysis, as demonstrated by SGFuzz [7], reinforce the strengths of the black box approach, which interactively captures complex state transitions including flow control. Our method not only overcomes the limitations of static analysis but also enhances the understanding of HTTP/2 vulnerabilities under practical attack scenarios.

4 Proposed Framework for HTTP/2 Stateful Fuzzing

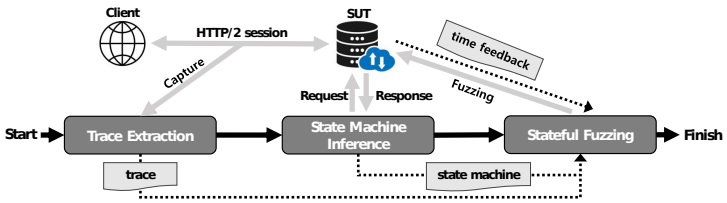
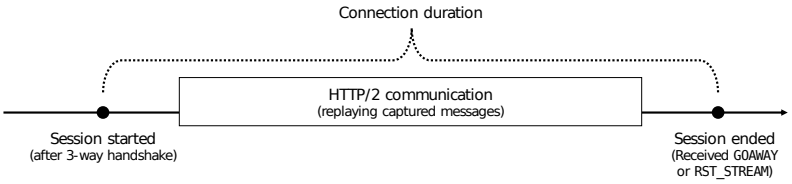
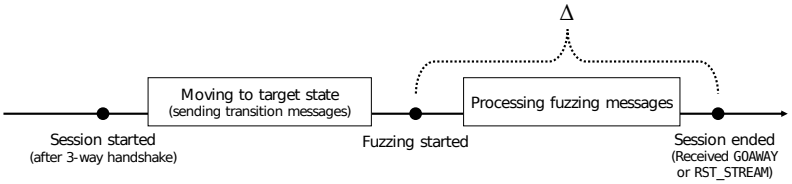


Fig. 3: PRETT2 framework

This section describes PRETT2, a stateful fuzzing framework designed for the automated detection of DoS vulnerabilities within diverse HTTP/2 implementations. Figure 3 illustrates the comprehensive methodology employed by



(a) Connection duration of normal HTTP/2 communication



(b) Connection duration of HTTP/2 communication under fuzzing

Fig. 4: Connection duration for time feedback

PRETT2, which is designed to effectively identify and exploit vulnerabilities in HTTP/2 implementations through a three-phase process: trace extraction, state machine inference, and stateful fuzzing.

Improvements over PRETT. PRETT2 significantly advances beyond its predecessor, by tailoring its algorithm to address the specific challenges posed by the traits of HTTP/2. The following are several key distinctions between them:

- ***Message complexity:*** PRETT2 reflects both the binary frames and the order of messages in multiple streams. This differs from creating test cases by merging human-readable keywords and using a single connection.
- ***Data sources:*** PRETT2 analyzes protocol behavior through network traces alone, capturing both request and response times, without binary analysis.
- ***Connection duration:*** PRETT2 records the normal *connection duration* during state machine inference. It captures the timeframe from the start to the end of an HTTP/2 session (as shown in Figure 4a) and uses this for its subsequent stateful fuzzing.

4.1 Trace Extraction

PRETT2 employs network traces recorded between a server under test (SUT) and a client for state machine inference and stateful fuzzing, using these traces as templates to generate fuzzing messages. By generating HTTP/2 traffic, the inherent feature of stream multiplexing is recorded, serving as a basis for the state machine inference that incorporates flow control and for the multiplexed fuzzing initiatives.

To efficiently capture these traces, an automated script can be utilized, combining the use of packet capture software like Wireshark with tshark command-line interface and a browser command for initiating traffic. An example of an

automated capture script is shown in Listing 2. This script encapsulates the process of initiating the capture of HTTP/2 traffic using tshark, along with the concurrent initiation of a Chrome session directed to a specified HTTP/2-enabled server.

Listing 2: An example script for automated HTTP/2 traffic capture

```
$ tshark -i <interface> -Y "http2" -w captured_traffic.pcap &
  chrome --headless --disable-gpu --dump-dom https://<server_ip>
```

Captured traces are encrypted using the TLS protocol version 1.2 or higher for security reasons [45], limiting access to message details and complicating frame parameter manipulation for fuzzing. Thus, they need to be decrypted using a keylog file [44].

4.2 State Machine Inference

Given network traces, PRETT2 uses an interactive method of protocol reverse engineering that uses a request-and-response approach to infer state machines. PRETT2 utilizes the state machine inference algorithm presented in a previous study, PRETT [28], by repeating both state machine expansion and state machine minimization processes for each level based on a tree model.

State Machine Expansion. PRETT2 constructs a state machine based on a tree structure, in which the root node is considered an initial state. Assuming the maximum level of the tree is L, PRETT2 expands the tree to have as many nodes in the next level $L + 1$ as possible by sending candidate messages on every leaf node. The candidate messages in the traces are presented as request messages without modification to their parameters. This prevents early rejection of malformed messages by the SUT. PRETT2 then checks the response messages with their response time and regards all the nodes expanded from the leaves as candidate nodes.

State Machine Minimization. After the state machine expansion is complete, PRETT2 eliminates duplicate candidate nodes in level $L + 1$ of the tree. To remove the duplicated nodes, PRETT2 conducts the compatibility test presented in the previous study [28]. The compatibility test finds redundant candidate nodes by determining either the compatibility or the incompatibility between nodes. In PRETT2, each candidate node is tested with other *valid* nodes based on a set of response time, request message, and response message. The valid nodes are the nodes proven to be states after the same process is conducted in the previous level.

End Condition. After all the candidate nodes are tested for compatibility with the other valid nodes, PRETT2 regards the candidate nodes that are proved to be unique (i.e., incompatible with any of the valid nodes) as a set of valid states

found in the level $L + 1$. Each valid state is used as a leaf node for the state expansion of the next level. In particular, when no valid state is present as a result of state minimization in the level, no state expansion and minimization process needs to be performed and a minimized state machine that is composed of only valid states is returned.

4.3 Stateful Fuzzing

Given the state machine inferred in the previous step, PRETT2 generates test cases based on the request messages of traces. Specifically, PRETT2 mutates the variable fields of the HTTP/2 frames that exist in a message, considering the HTTP/2 stream multiplexing feature by simulating multiple interleaved streams in a single connection. Numerous test cases are transmitted to the SUT by traversing every transition in the state machine. Then PRETT2 takes time feedback from SUT in return, which is a new connection duration Δ (as shown in Figure 4b). To determine the potential exhaustion of the connection pool, PRETT2 compares the connection duration Δ to that of normal HTTP/2 communication recorded in the previous step.

5 Evaluation

This section presents the experimental outcomes of protocol reverse engineering and stateful fuzzing carried out with PRETT2. We utilized network traces obtained from multiple web servers and browsers to ensure diversity, taking into account their popularity[2]. PRETT2 leveraged these traces to conduct state machine inference and stateful fuzzing on the web servers. To ensure the reproducibility of our results, we excluded commercial implementations such as Cloudflare Server. For the reproducibility of our work, we published the source code for both the state machine inference and stateful fuzzing to the public [3].

5.1 Experimental Setup

We ran a series of experiments involving protocol reverse engineering and stateful fuzzing on VMware virtual machines. Each virtual machine was configured identically, equipped with an Intel i5 CPU that has 12 logical cores running at 2.90GHz, Ubuntu 18.04.6 LTS, 40GB of SSD, and 4GB of system memory. We captured network traces between a server and a client by configuring each server with the default settings for HTTP/2 and requesting an index page and a favicon from each client. Every server was set up so that resources for the index page were available, but those for the favicon were unavailable. Different HTTP/2 implementations were installed on each virtual machine to examine their behavior under identical conditions.

[2] For the usage share of HTTP/2 web servers and browsers, we referred to the statistics by W3Techs [52] and W3Schools [51], respectively.

[3] https://github.com/choonginlee/PRETT2

Table 4: Targets of HTTP/2 implementation

Servers				**Browsers**			
	Target	Release	Announced		Target	Release	Announced
Nginx	v1	1.14.0	06-Apr-18	Chromium	v1	62	31-Aug-17
	v2	1.21.6	25-Jan-22		v2	67	29-May-18
	latest	1.23.4	28-Mar-23		latest	111	01-Mar-23
Apache	v1	2.4.29	23-Oct-17	Firefox	v1	56	28-Sep-17
	v2	2.4.33	28-Mar-18		v2	60	09-May-18
	latest	2.4.56	07-Mar-23		latest	111	14-Mar-23
H2O	v1	2.2.4	15-Dec-17	Opera	v1	48	27-Sep-17
	v2	2.3.0-beta1	02-Jun-18		v2	53	10-May-18
	latest	2.3.0-beta2	14-Aug-19		latest	97	22-Mar-23

Target Selection. We assessed three HTTP/2 web servers - Nginx, Apache, and H2O. Nginx [19] is the most widely used web server for HTTP/2, accounting for approximately 34% of usage on millions of websites. Apache, also known as httpd [5], is the second most widely used, representing around 30% of usage. H2O [18] is used by fewer websites, but it is commonly employed by high-traffic sites. In contrast, we evaluated three HTTP/2 web browsers - Chromium [42], Firefox [31], and Opera [34]. We chose to test the versatility of state machine inference by arbitrarily selecting outdated versions (v1, v2) as well as the latest version [4] of each implementation. We excluded extremely outdated versions that are not supported by the operating system. The selected HTTP/2 implementations are summarized in Table 4.

Control Variables. To minimize external factors that could affect the experiment's outcome, we carefully configured the environment. First, we selected an operating system that supports all the various versions of multiple HTTP/2 implementations without switching kernels. Second, we experimented with a closed local area network (LAN) to prevent any potential interference from other network services. Third, we configured the HTTP/2 web servers to support only IPv4 and not to prefetch assets requested by a client. Finally, we ensured that the resources for the index page of each server were identical.

5.2 State Machine Inference

During our experiments, we recorded a total of 81 traces by combining nine servers and nine web browsers. Using PRETT2, we were able to successfully infer multiple state machines that effectively demonstrate the flow control process depending on three variables: server type, client type, and server version. We have demonstrated the representative state machines for Nginx and H2O in Figure 5 and Figure 6. For the Apache server, PRETT2 inferred different state

[4] We determined the latest version of each implementation based on the official release at the time of the evaluation in March 2023.

machines for each server-browser combination. The state machines for Apache are more sophisticated than those for Nginx and H2O. Due to the page limitation, we have provided an example of an Apache state machine in the Appendix.

Results. We represent state machines from a client point of view in a directional graph. The edges specify a pair of sent-received messages (split by a slash) with one or more HTTP/2 frames, while the nodes specify the state. The dots(...) in the received messages indicate that no message has been received from a server. All states except the END state are subject to change to the END state after receiving a `GOAWAY` frame, which occurs when it reaches the pre-configured time-out. Numbers in square brackets indicate each stream ID on which its message is sent, and a bar(|) is used to differentiate multiple frames of a message in separate streams. To make it more readable, we shortened the name of each frame into two alphabetical letters [5].

Analysis. Figure 5 illustrates a state machine that was inferred for Nginx and H2O servers. This was done using traces that recorded communication with Chromium and Opera browsers. On every state except the END state, both the client and server could configure the window size using `SETTINGS` and `WINDOW_UPDATE` frames. States DATA1 and DATA2 represent the states in which the client received and responded to a request for the first data (i.e., the index page) and second data (i.e., the favicon), respectively. It is noteworthy that data requests using different types of `HEADER` frames make different transitions on each state. For instance, servers responded quickly with a `GOAWAY` frame if the second data was requested on the INIT state. However, they responded with both `HEADER` and `DATA` frames when requested the same request on the DATA state. Interestingly, we observed slightly different responses to the request for the second data in state DATA1 when using the v1 version of H2O. This was due to the different sizes of header block fragments, which varied depending on the versions.

Figure 6 shows three different state machines inferred for either H2O or Nginx. This was done using traces that recorded communication with Firefox browsers. These state machines differ from the one shown in Figure 5 due to the message difference in the traces. Three significant differences compared to other

[5] `ST`: `SETTINGS`, `WU`: `WINDOW_UPDATE`, `HD`: `HEADERS`, `DA`: `DATA`, `GO`: `GOAWAY`, `PR`: `PRIORITY`, and `RS`: `RST_STREAM`

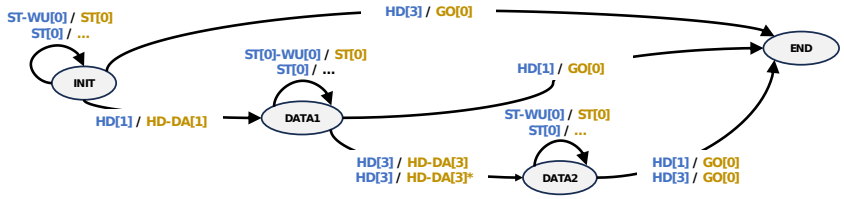
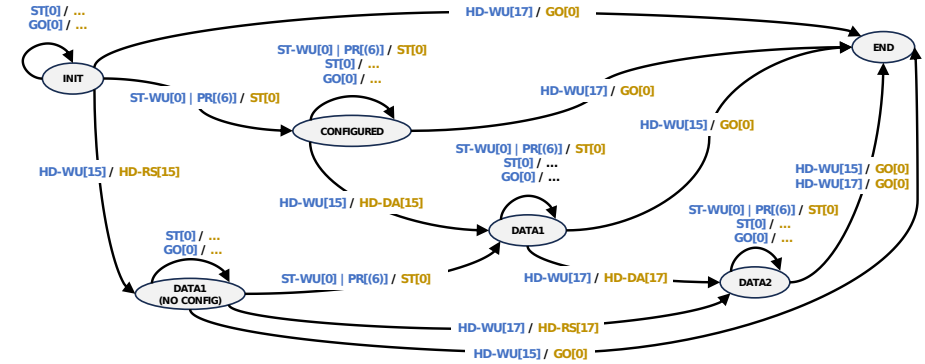
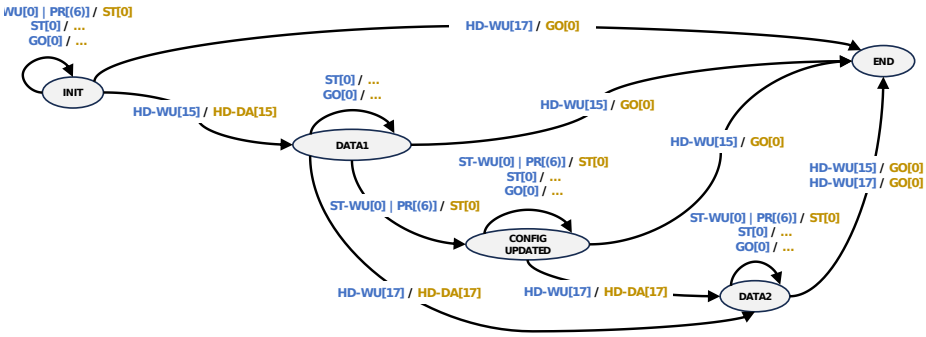


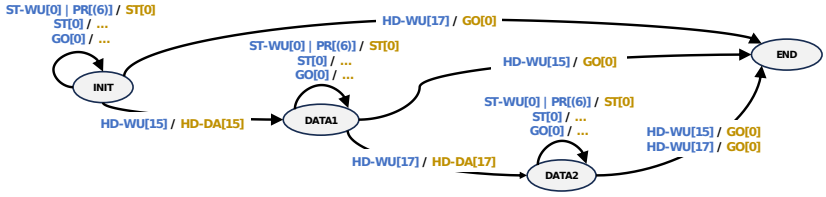
Fig. 5: State machine of Nginx and H2O (Chromium and Opera traces)



(a) State machine of H2O (Firefox trace)



(b) State machine of Nginx (version v2 and latest, Firefox trace)



(c) State machine of Nginx (version v1, Firefox trace)

Fig. 6: State machines of H2O and Nginx (Firefox traces)

browsers are as follows. (1) Firefox reserved idle streams [54] for stream prioritization using `PRIORITY` frames[6]. (2) Firefox requested data using two frames, such as `HEADER` and `WINDOW_UPDATE`, in the same message. (3) Firefox sent a `GOAWAY` frame back to the server when it had received a `GOAWAY` frame.

H2O and Nginx worked differently from each other for the communication with the Firefox browser. For instance, Figure 6a shows that H2O had a CON-

[6] The `PRIORITY` frame reserved a total of six streams with stream IDs 3, 5, 7, 9, 11, and 13. This short frame is shown as PR[(6)] in the state machine. As a result, the first and second data requests were on stream IDs 15 and 17, respectively.

FIGURED state indicating that a server was requested for the configuration settings via `SETTINGS`, `WINDOW_UPDATE`, and `PRIORITY` frames for the first time. Additionally, H2O sent an `RST_STREAM` frame whenever a client requested data without such configuration settings. These differences were not seen in the case of Nginx (Figure 6b). We could observe that the v1 version of H2O also worked differently (Figure 6c) compared to other versions (Figure 6b.) The state machine was the same as that of communication with Chromium and Opera, except for several additional messages originating from Firefox messages.

Comparison to SGFuzz. To our knowledge, SGFuzz [7] is the only work that has demonstrated the ability to perform state machine inference for fuzzing. They presented a compact state machine for H2O using a state transition tree (STT), which represents the lifecycle of an HTTP/2 stream. To generate the state transition tree, SGFuzz refers to state variables that are assigned and compared to named constants using static resources of the HTTP/2 implementation. The state machine is inferred effectively because it contains specific information on the states that detail those in the HTTP/2 specification.

However, they did not successfully demonstrate flow control and multiplexed streams in practice: the state machines lack any information on configuration settings. On the other hand, PRETT2 is capable of inferring various state machines that demonstrate flow control messages on multiple streams. Specifically, the state machines contain the process of configuration settings via `SETTINGS` or `WINDOW_UPDATE`, which is done on stream ID 0, whereas requests for data are done on different stream IDs.

5.3 Security Analysis

As a result of fuzzing HTTP/2 servers using inferred state machines, PRETT2 could discover three Denial-of-Service attacks that exploit flow control flaws in both Apache HTTP servers (versions 2.4.46 and 2.4.58) and Nginx (version 1.25.3). The attacks were successful by sending multiple messages that exceeded the limit of the connection pool. We reported all the flaws to the developers and found that the attacks could be exploited when the server was configured by default without any external mitigation options.

Case Study. As a case study, we explain a vulnerability that we discovered via PRETT2. The vulnerability, known as CVE-2023-43622, was triggered by an implementation flaw in flow control for Apache httpd 2.4.58. To discover this vulnerability, we used PRETT2 to craft parameters of HTTP/2 frames with special values by fuzzing and assessing each state. Specifically, we sent a `SETTINGS` frame with a 0 `SETTINGS_INITIAL_WINDOW_SIZE` value and sent hundreds of data requests by traversing every possible state transition based on the state machine. This action fooled the server into believing that the client was not able to receive data and led to a slow-rate attack.

The attack was successful when we used a state machine based on communication traces between the server and the Chromium browser. However, we could not discover the vulnerability using the same parameters based on the communication traces with the Firefox browser. This is because Firefox sends `WINDOW_UPDATE` frames immediately after connection establishment, which convinces the web server that the client can receive the data. Instead, we were able to successfully exploit the vulnerability by manipulating the parameter `window_size_increment` of a `WINDOW_UPDATE` frame using fuzzed messages.

6 Related Work

HTTP/2 DoS. Various approaches have analyzed the security vulnerabilities of HTTP/2 DoS using manual analysis. Targeting the extended features of HTTP/2, H2DoS [29] observed that a malformed sequence of crafted frames can cause DoS. Imperva [24] tested the features of HTTP/2 and found several attack vectors for DoS. Also, HTTP/2 Distributed DoS (DDoS) attacks are presented by various literature. Adi et al. [3] presented an HTTP/2 DDoS attack model, where the server resources are exhausted with the large volume, but legitimate flash crowd traffic. Adi et al. [1] further discussed how to camouflage offending traffic, such as the traffic of the attack [3]. HTTP/2 Cannon [8] analyzed the HTTP/1.1 legacy functionalities of HTTP/2 and explored the DDoS risk posed by the upgraded functionality of the HTTP/2 protocol. HTTP/2 Tsunami [50] observed that an amplification DDoS attack is possible in a proxy environment through HPACK. The effects of a simple asymmetric DDoS attack on HTTP/1.1 and HTTP/2 servers are compared in another paper [38]. Although HTTP/2 performs better due to its performance improvements, it is found to be vulnerable to a new attack vector called the multiplexed asymmetric attack.

Slow-rate Attack. Security researchers focused on the slow-rate attack as a common attack for HTTP/2 DoS. Adi et al. [2] described how to launch several slow-rate attacks and investigated its effect. Tripathi et al. [48] proposed a feature-based statistical anomaly detection mechanism for the five types of attacks that they discovered, three of which are structurally comparable to known HTTP/1.1 attacks. Zhang et al. [59] also presented another type of slow-rate attack called the zAttack.

Protocol Reverse Engineering. Three types of protocol reverse engineering methods have been proposed for the inference of protocol communication. First, the static method clustered network traces to infer message formats and protocol states [4, 39, 46, 53]. For example, ReverX [4] generated a state machine through the process of merging and simplifying messages. Veritas [53] used statistical analysis by defining a probabilistic protocol state machine (P-PSM) which is a probabilistic generalization of the protocol state machine. Second, the dynamic method analyzed execution traces from implementations to understand communication rules and state machines [14, 56]. Xiao et al. [56] inferred an

accurate state machine using the trial-and-error learning method for messages. Prospex [14] clustered protocol message sequences and analyzed their correlations. Lastly, the interactive method used a request-and-response approach using network traces to determine the correctness of messages and their sequences to infer a state machine [13, 27, 60]. Cho et al. [13] and Zhang et al. [60] identified potential data message fields by sending and receiving messages as query strings.

HTTP/2 Fuzzing. Compared to the variety of methods for manual analysis, only a few automated testing methods (i.e., fuzzing) were presented. Http2fuzz [41] is the first fuzzer for testing HTTP/2 vulnerabilities presented by the Yahoo Pentest Team. SGFuzz [7] presented an advanced fuzzing framework using stateful information. As discussed earlier, however, they are limited to addressing challenges for discovering HTTP/2 DoS due to the insufficient testing of flow control.

7 Concluding Remarks

The widespread adoption of HTTP/2 services by web servers has led to a steady increase in DoS vulnerabilities due to implementation flaws in HTTP/2 flow control. As a remedy, we proposed PRETT2, a stateful fuzzing framework designed to automatically analyze HTTP/2 DoS flaws. PRETT2 infers state machines using a novel method for automated protocol reverse engineering, successfully deriving various types depending on server type, client type, and server version. Utilizing these state machines, PRETT2 conducted stateful fuzzing on multiple HTTP/2 servers and discovered several DoS attacks exploiting flow control flaws. Despite our promising results, PRETT2 has limitations: the heavy dependence on network traces for the state machine and its focus solely on HTTP/2 flow control flaws. For future work, we aim to extend our research to include HTTP/3 testing, adapting our framework to tackle the evolving challenges in newer protocol versions. We believe that our research can significantly contribute to network security by mitigating DoS attacks on HTTP protocols as a proactive analysis method.

Acknowledgments. Special thanks to Taewoo Kim, whose contributions during the initial phase of this project helped lay the groundwork for our research. The authors thank the anonymous reviewers for their valuable comments. This work was supported by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2022-0-00277, Development of SBOM Technologies for Securing Software Supply Chains, No. 2022-0-01198, Convergence Security Core Talent Training Business (Korea University), and IITP-2024-2020-0-01819, ICT Creative Consilience program), PSC/CUNY Cycle 51 Software Security grant, and the Research Foundation of the City University of New York (RFCUNY).

Appendix

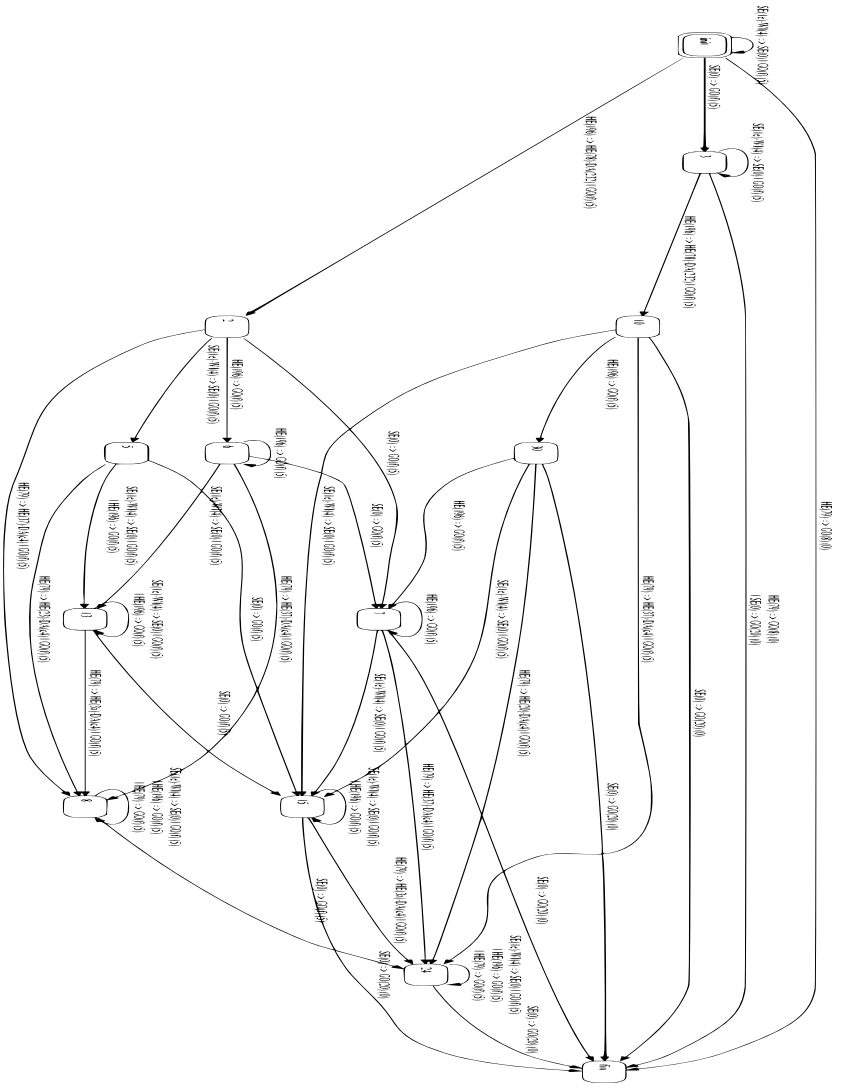


Fig. 7: The illustration of the state machine for Apache's latest version, inferred using Chromium traces. While the detailed representation is optimized for a broad overview, it highlights the intricate complexity and extensive capabilities of Apache in contrast to the more simplified state machines of H2O and Nginx.

References

1. Adi, E., Baig, Z., Hingston, P.: Stealthy Denial of Service (DoS) attack modelling and detection for HTTP/2 services. Journal of Network and Computer Applications **91**, 1–13 (2017)
2. Adi, E., Baig, Z., Lam, C.P., Hingston, P.: Low-rate denial-of-service attacks against HTTP/2 services. In: 5th IEEE International Conference on IT Convergence and Security (ICITCS). pp. 1–5 (2015)
3. Adi, E., Baig, Z.A., Hingston, P., Lam, C.P.: Distributed denial-of-service attacks against HTTP/2 services. Cluster Computing **19**, 79–86 (2016)
4. Antunes, J., Neves, N., Verissimo, P.: Reverse engineering of protocols from network traces. In: 18th Working Conference on Reverse Engineering. pp. 169–178. IEEE (2011)
5. Apache Software Foundation: Apache HTTP Server. `https://httpd.apache.org/`
6. Athapathu, R.: HTTP/2 Flow Control (2019), `https://medium.com/coderscorner/http-2-flow-control-77e54f7fd518`
7. Ba, J., Böhme, M., Mirzamomen, Z., Roychoudhury, A.: Stateful Greybox Fuzzing. In: 31st USENIX Security Symposium. pp. 3255–3272 (2022)
8. Beckett, D., Sezer, S.: HTTP/2 Cannon: Experimental analysis on HTTP/1 and HTTP/2 request flood DDoS attacks. In: Seventh IEEE International Conference on Emerging Security Technologies (EST). pp. 108–113 (2017)
9. Belson, D., Pardue, L.: Examining HTTP/3 usage one year on. Cloudflare (2023), `https://blog.cloudflare.com/http3-usage-one-year-on/`
10. Berners-Lee, T., Fielding, R., Frystyk, H.: RFC1945: Hypertext transfer protocol–http/1.0 (1996)
11. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In: 14th ACM conference on Computer and Communications Security. pp. 317–329 (2007)
12. Cambiaso, E., Papaleo, G., Chiola, G., Aiello, M.: Slow DoS attacks: definition and categorisation. International Journal of Trust Management in Computing and Communications **1**, 300–319 (2013)
13. Cho, C.Y., Babi ć, D., Shin, E.C.R., Song, D.: Inference and Analysis of Formal Models of Botnet Command and Control Protocols. In: 17th ACM conference on Computer and Communications Security. pp. 426–439 (2010)
14. Comparetti, P.M., Wondracek, G., Kruegel, C., Kirda, E.: Prospex: Protocol Specification Extraction. In: 30th IEEE Symposium on Security and Privacy. pp. 110–125 (2009)
15. Corbel, R., Stephan, E., Omnes, N.: HTTP/1.1 pipelining vs HTTP2 in-the-clear: Performance comparison. In: 13th IEEE International Conference on New Technologies for Distributed Systems (NOTERE). pp. 1–6 (2016)
16. CVE-2016-1546. National Vulnerability Database (Jul 2016), `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2016-1546`
17. De Ruiter, J., Poll, E.: Protocol State Fuzzing of TLS Implementations. In: 24th USENIX Security Symposium. pp. 193–206 (2015)
18. DeNA: H2O Web Server. `https://h2o.examp1e.net/`
19. F5: Nginx. `https://nginx.org/`
20. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol–http/1.1. Tech. rep. (1999)
21. Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K.: Pulsar: Stateful black-box fuzzing of proprietary network protocols. In: International Conference on Security and Privacy in Communication Systems. pp. 330–347 (2015)

22. Guo, R., Li, W., Liu, B., Hao, S., Zhang, J., Duan, H., Shen, K., Chen, J., Liu, Y.: CDN Judo: Breaking the CDN DoS Protection with Itself. In: Network and Distributed System Security Symposium (2020)
23. IETF HTTP Working Group: List of known HTTP/2 implementations (2020), https://github.com/httpwg/http2-spec/wiki/Implementations
24. Imperva: HTTP/2: In-depth analysis of the top four flaws of the next generation web protocol (2017), https://www.imperva.com/docs/Imperva_HII_HTTP2.pdf
25. Jiang, M., Luo, X., Miu, T., Hu, S., Rao, W.: Are HTTP/2 servers ready yet? In: 37th IEEE International Conference on Distributed Computing Systems (ICDCS). pp. 1661–1671 (2017)
26. Kaloper-Meršinjak, D., Mehnert, H., Madhavapeddy, A., Sewell, P.: Not-Quite-So-Broken TLS: Lessons in re-engineering a security protocol specification and implementation. In: 24th USENIX Security Symposium. pp. 223–238 (2015)
27. LaRoche, P., Burrows, A., Zincir-Heywood, A.N.: How far an evolutionary approach can go for protocol state analysis and discovery. In: IEEE Congress on Evolutionary Computation. pp. 3228–3235 (2013)
28. Lee, C., Bae, J., Lee, H.: PRETT: Protocol Reverse Engineering Using Binary Tokens and Network Traces. In: IFIP International Conference on ICT Systems Security and Privacy Protection. pp. 141–155 (2018)
29. Ling, X., Wu, C., Ji, S., Han, M.: H_2 DoS: An Application-Layer DoS Attack Towards HTTP/2 Protocol. In: International Conference on Security and Privacy in Communication Systems. pp. 550–570 (2017)
30. Mirković, J., Dietrich, S., Dittrich, D., Reiher, P.: Internet Denial of Service: Attack and Defense Mechanisms. Prentice Hall PTR (2004)
31. Mozilla Foundation: Mozilla Firefox. https://www.mozilla.org/firefox/
32. Narayan, J., Shukla, S.K., Clancy, T.C.: A survey of automatic protocol reverse engineering tools. ACM Computing Surveys (CSUR) **48**, 1–26 (2015)
33. Nkuba, C.K., Kim, S., Dietrich, S., Lee, H.: Riding the iot wave with vfuzz: discovering security flaws in smart homes. IEEE Access **10**, 1775–1789 (2021)
34. Opera Software: Opera Browser. https://www.opera.com/
35. Park, H., Nkuba, C.K., Woo, S., Lee, H.: L2fuzz: Discovering bluetooth l2cap vulnerabilities using stateful fuzz testing. In: 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 343–354 (2022)
36. Pham, V.T., Böhme, M., Roychoudhury, A.: AFLNET: A Greybox Fuzzer for Network Protocols. In: IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). pp. 460–465 (2020)
37. Poll, E., De Ruiter, J., Schubert, A.: Protocol state machines and session languages: specification, implementation, and security flaws. In: IEEE Security and Privacy Workshops. pp. 125–133 (2015)
38. Praseed, A., Thilagam, P.S.: Multiplexed asymmetric attacks: Next-generation DDoS on HTTP/2 servers. IEEE Transactions on Information Forensics and Security **15**, 1790–1800 (2019)
39. Shevertalov, M., Mancoridis, S.: A reverse engineering tool for extracting protocols of networked applications. In: 14th Working Conference on Reverse Engineering (WCRE). pp. 229–238 (2007)
40. Stenberg, D.: HTTP2 Explained. ACM SIGCOMM Computer Communication Review **44**, 120–128 (2014)
41. Stuart Larsen, J.V.: Attacking HTTP/2 Implementations. Yahoo pentest team (2015), https://www.slideshare.net/JohnVillamil/attacking-http2-implementations-1

42. The Chromium Projects: Chromium. https://www.chromium.org/
43. The Chromium Projects: SPDY: An experimental protocol for a faster web (2009), https://www.chromium.org/spdy/
44. Thomson, M.: The SSLKEYLOGFILE Format for TLS. Tech. rep., Internet Engineering Task Force (Apr 2024), https://datatracker.ietf.org/doc/draft-ietf-tls-keylogfile/01/
45. Thomson, M., Benfield, C.: HTTP/2. RFC 9113 (Jun 2022). https://doi.org/10.17487/RFC9113, https://www.rfc-editor.org/info/rfc9113
46. Trifilò, A., Burschka, S., Biersack, E.: Traffic to protocol reverse engineering. In: IEEE Symposium on Computational Intelligence for Security and Defense Applications. pp. 1–8 (2009)
47. Tripathi, N.: Delays have Dangerous Ends: Slow HTTP/2 DoS attacks into the Wild and their Real-Time Detection using Event Sequence Analysis. IEEE Transactions on Dependable and Secure Computing (2023)
48. Tripathi, N., Hubballi, N.: Slow rate denial of service attacks against HTTP/2 and detection. Computers & Security **72**, 255–272 (2018)
49. Tripathi, N., Shaji, A.K.: Defer No Time, Delays have Dangerous Ends: Slow HTTP/2 DoS Attacks into the Wild. In: 14th IEEE International Conference on COMmunication Systems & NETworkS (COMSNETS). pp. 194–198 (2022)
50. Beckett, D., Sezer, S.: HTTP/2 Tsunami: Investigating HTTP/2 proxy amplification DDoS attacks. In: Seventh IEEE International Conference on Emerging Security Technologies (EST). pp. 128–133 (2017)
51. W3Schools: The most popular browsers (Jan 2024), https://www.w3schools.com/browsers
52. W3Techs: Web servers market position report (Jan 2024), https://w3techs.com/technologies/market/web_server
53. Wang, Y., Zhang, Z., Yao, D.D., Qu, B., Guo, L.: Inferring Protocol State Machine from Network Traces: A Probabilistic Approach. In: International Conference on Applied Cryptography and Network Security. pp. 1–18 (2011)
54. Wijnants, M., Marx, R., Quax, P., Lamotte, W.: HTTP/2 Prioritization and its Impact on Web Performance. In: Proceedings of the 2018 World Wide Web Conference. pp. 1755–1764 (2018)
55. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic network protocol analysis. In: Network and Distributed System Security Symposium (2008)
56. Xiao, M.M., Yu, S.Z., Wang, Y.: Automatic network protocol automaton extraction. In: Third International Conference on Network and System Security. pp. 336–343 (2009)
57. Yahia, M.B., Le Louedec, Y., Simon, G., Nuaymi, L.: HTTP/2-Based Streaming Solutions for Tiled Omnidirectional Videos. In: IEEE International Symposium on Multimedia (ISM). pp. 89–96 (2018)
58. Yahia, M.B., Louedec, Y.L., Simon, G., Nuaymi, L., Corbillon, X.: HTTP/2-based Frame Discarding for Low-Latency Adaptive Video Streaming. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM) **15**, 1–23 (2019)
59. Zhang, Y., Shi, Y.: A Slow Rate Denial-of-Service Attack Against HTTP/2. In: IEEE 4th International Conference on Computer and Communications (ICCC). pp. 1388–1391 (2018)
60. Zhang, Z., Wen, Q.Y., Tang, W.: Mining Protocol State Machines by Interactive Grammar Inference. In: Third IEEE International Conference on Digital Manufacturing & Automation. pp. 524–527 (2012)