Delays Have Dangerous Ends: Slow HTTP/2 DoS Attacks Into the Wild and Their Real-Time Detection Using Event Sequence Analysis

Nikhil Tripathi, Member, IEEE

Abstract—Jon Postel’s robustness principle states that the communicating entities should be liberal while accepting the data. Several web servers on the Internet do follow this principle as they wait to receive the remaining portion of an incomplete web request. Unfortunately, this behaviour also makes them vulnerable to Slow Rate DoS attacks. A few approaches are known to counter Slow Rate DoS attacks against HTTP/1.1. However, those defence approaches are incompatible with HTTP/2 because of several operation differences between HTTP/1.1 and its successor, HTTP/2. Also, to the best of our knowledge, no defence scheme is known to detect Slow Rate DoS attacks against an HTTP/2 supporting web server in real-time. To bridge this gap, in this article, we propose an event sequence analysis-based scheme to detect Slow HTTP/2 DoS attacks. Using extensive experiments, we show that the scheme can detect attacks in real-time with high accuracy and marginal computational overhead. As an aside, we also present a study on the behaviour of popular HTTP/2 servers on the Internet against Slow HTTP/2 DoS attacks. Surprisingly, we noticed that several of them are vulnerable to these attacks, thereby justifying the requirement for an effective real-time detection strategy.

Index Terms—Anomaly detection, event sequence analysis, HTTP/2, slow rate DoS attacks.

I. INTRODUCTION

The Internet is known to host various services today, such as web, mail, file transfer, etc. These services exchange messages using different application layer protocols. Some of the most popular application layer protocols that are being used over the Internet today are HyperText Transfer Protocol (HTTP), Network Time Protocol (NTP), Domain Name System (DNS), File Transfer Protocol (FTP), and Simple Mail Transfer Protocol (SMTP). These protocols are vulnerable to a class of attacks commonly known as application layer Denial-of-Service (DoS) attacks [1], [2], [3], [4], [5], [6]. These are a newer class of DoS attacks that exploit vulnerabilities in the application layer protocols. These attacks target specific services (e.g., web, mail, file transfer) running on a computer [7], [8], [9]. Application layer DoS attacks can disrupt a service running on a computationally powerful server using minimal resources [2]. These attacks generate a smaller amount of traffic than network/transport layer Distributed DoS (DDoS) attacks, due to which they are known to be relatively stealthier [1], [2]. Mitigating potential application layer DoS attack vulnerabilities in a protocol requires modification in the protocol’s specifications. However, modifying a protocol specification is cumbersome, requiring discussions and deliberations between stakeholders at different levels, which take considerable time. Moreover, applying these changes to different protocol implementations and releasing their newer versions by the implementation vendors also take a substantial amount of time [2]. Due to these reasons, application layer DoS attacks are real threats to the organisational assets available over the Internet today.

In the last decade, security researchers disclosed vulnerabilities in several application layer protocols and their software implementations. These vulnerabilities can be exploited to launch different types of DoS attacks. HTTP is considered one of the most studied protocols against application layer DoS attacks [2]. The vulnerabilities related to HTTP can be classified into three different categories - 1) implementation-specific vulnerabilities, 2) flood-based application layer DDoS attacks, and 3) Slow Rate DoS attacks. The vulnerabilities disclosed in the HTTP implementations (e.g., Apache, IIS, etc.) were easier to mitigate as the corresponding vendors patched them post their exposure [10]. Also, the flood-based application layer DDoS attacks that require sending many requests to the victim web server were difficult to launch and easier to detect [11]. However, Slow Rate DoS attacks [7], [12] need small computational power to execute on the attacker’s side. They involve sending very few requests to the web server to prevent it from serving benign clients [2], [11]. Since these attacks generate less traffic, they are stealthier and, thus, difficult to detect compared to flood-based application layer DDoS attacks [2], [11], [13]. Taking motivation from this, we restrict our focus to Slow Rate DoS attacks in this paper.

Slow Rate DoS attack requires sending multiple incomplete requests to the victim server to consume its connection queue space. Once this finite space is consumed, the server can no longer entertain incoming requests. A few instances of Slow Rate DoS attacks have also been encountered in the past [2], [14]. Several known Slow Rate DoS attacks against HTTP/1.1 [2] exist. However, due to several operational differences between HTTP/1.1 and its successor, HTTP/2, the known Slow Rate DoS attacks against HTTP/1.1 are ineffective against HTTP/2. Likewise, the defence mechanisms known to counter Slow HTTP/1.1 DoS attacks [2] can not be deployed to detect these...
attacks against HTTP/2. In one of our recent works [12], we presented a few novel Slow Rate DoS attacks that are effective against HTTP/2 and proposed a defence mechanism to counter the attacks. However, the attacks were tested in a controlled lab setup only, and the defence approach had a few limitations (e.g. no real-time detection, attack rate dependency, etc.). These gaps in the study of HTTP/2 behaviour against Slow Rate DoS attacks motivated us to 1) test the impact of Slow HTTP/2 DoS attacks on HTTP/2 supporting web servers on the Internet and 2) propose a defence mechanism to detect the attacks proactively. We make the following specific contributions in this paper:

- We test the behaviour of HTTP/2 supporting web servers on the Internet against different Slow HTTP/2 DoS attacks and show that several are vulnerable to attacks.
- We propose an event sequence analysis-based detection scheme to detect Slow HTTP/2 DoS attacks in real-time.
- We test the detection performance of the proposed scheme in a real network and show that it can detect attacks with very high accuracy and marginal computational overhead.
- We compare the detection performance of the proposed scheme with the previously known defence mechanisms to counter Slow HTTP/2 DoS attacks and show that it outperforms the previously known defence mechanisms.

The rest of the paper is organised as follows. We present the HTTP/2 usage over the Internet and the literature review in Section II. Section III presents an empirical evaluation of Slow HTTP/2 DoS attacks. In Section IV, we discuss the proposed detection scheme. We present the experiments conducted to test the detection performance of the proposed method in Section V. Finally, the paper is concluded in Section VI.

II. BACKGROUND

In this section, we first present a brief overview of HTTP/2 and its usage on the Internet. Subsequently, we discuss the Slow Rate DoS attacks and known defence mechanisms.

A. HTTP/2 and Its Usage on the Internet

HTTP/2 is defined in Request For Comment (RFC) 7540 [15] and recently became a standard in May 2015. HTTP/2 operation differs significantly from its predecessor HTTP/1.1 [16], [17]. HTTP/2 was proposed considering the inability of HTTP/1.1 to efficiently utilise the TCP’s transmission capacity. HTTP/2 achieves higher efficiency using several virtual connections known as streams over a single TCP connection. HTTP/2 also mitigates head-of-line blocking of the web requests that persist in HTTP/1.1. HTTP/2 also implements its own flow control mechanism at the application layer. This mechanism is required to prevent the communicating entities from overwhelming each other by sending data simultaneously through multiple streams over a single TCP connection. HTTP/2 also involves using different types of frames for different purposes [15]. These frames are carried inside the TCP payload and should not be confused with the link-layer frames. Interested readers are requested to refer to RFC 7540 [15] for the list of HTTP/2 frames and their usage.

HTTP/2 Usage: We conducted a comprehensive study to measure HTTP/2 usage on the Internet. We took into account the top 500,000 Alexa domains in this study and tested if they support HTTP/2 over TLS (known as h2) or HTTP/2 over cleartext (known as h2c). We conducted this measurement study in four iterations. In the first iteration, we collected HTTP/2 usage statistics by performing TLS handshakes with the web servers of the top 500 K domains in the Alexa list. In this iteration, we received responses from 445315 web servers, of which 335854 web servers were found to support HTTP/2. Moreover, 109461 web servers were found to support HTTP/1.1. The remaining 54685 web servers did not respond in the first iteration. In the second iteration, we appended the www keyword to all those domains (=164146) whose web servers either supported HTTP/1.1 (=109461) or did not respond (=54685) in the first iteration. Subsequently, we collected HTTP/2 usage statistics by performing TLS handshakes with the web servers corresponding to these domains. In the third iteration, we measured the usage of h2c on the Internet using the HTTP scheme. We checked the HTTP/2 support only for those domains which either supported HTTP/1.1 or did not respond in the second iteration. In the fourth iteration, we appended the www keyword to the unresponsive domains in the third iteration. Subsequently, we collected HTTP/2 statistics by sending cleartext HTTP/2 web requests to the web servers corresponding to these domains. The measurement results for each iteration are shown in Table I. It can be noticed from the table that the web servers corresponding to 356417 (sum of the numbers shown in bold fonts in Table I) domains were found to support HTTP/2.

B. Related Works

Several works in the literature discuss DoS/DDoS attacks against HTTP/1.1 and the defence mechanisms to counter those attacks [1], [2], [13]. However, the same does not hold for its successor, HTTP/2, as it is a recently standardized protocol [2]. Since HTTP/2 operation differs significantly from HTTP/1.1 [16], [17], the existing techniques to detect DoS attacks against HTTP/1.1 [7], [18], [19], [20], [21] are ineffective in detecting DoS attacks against HTTP/2. As a result, researchers in the security community recently proposed a few defence approaches to detect DoS attacks against HTTP/2. This section first mentions DoS/DDoS attacks known against HTTP/2 and, subsequently, the defence mechanisms to counter them.

1) DoS/DDoS Attacks Against HTTP/2: Imperva [10] disclosed some implementation vulnerabilities in HTTP/2, which can be exploited to create situations such as a Blue Screen of Death and arbitrary code execution at the client side. However, these vulnerabilities existed in the protocol implementations but not in the protocol itself. Post the vulnerability exposure, the affected implementations were patched to mitigate these vulnerabilities. Beckett and Sezer [16], [17] studied how HTTP/2 functionalities can be exploited to launch flood and amplification-based DDoS attacks. Adi et al. [22] tested how the HTTP/2 web servers behave if control frame flooding is launched against them. Praseed and Thilagam [23], [24] demonstrated that the request multiplexing feature of HTTP/2 can be
exploited to launch asymmetric DDoS attacks on web servers. In particular, the attacker sends high-workload web requests resulting in heavy computational overhead at the web server. The multiplexed asymmetric attack may result in either partial DoS, also known as Reduction-of-Quality (RoQ), or a complete DoS scenario. The attacks discussed in these works are distributed in fashion and, thus, typically fall under the Distributed Denial-of-Service (DDoS) category [2]. Since these attacks require sending many web requests to the server to render it unusable, they are relatively difficult to launch and easier to detect [11]. Slow HTTP/2 DoS attacks [13] require sending multiple specially crafted incomplete requests to a victim server. On receiving such requests, the server stores them in its connection queue space and waits for a time duration \( T \) (predefined in the configuration file) in the hope of receiving the remaining portions of the web request. However, the attacker never sends the remaining portion of the complete requests. To consume the server’s connection queue space, the attacker establishes enough connections, and from each connection, it sends incomplete requests. Once the connection queue space is consumed, the server can not entertain further incoming requests, resulting in a DoS scenario. To craft incomplete web requests, an attacker modifies different parameters in the HTTP/2 frames. Based on the altered parameter, there can be five variants of Slow HTTP/2 DoS attacks\(^1\) - 1) Advertising Zero Window Size (Attack-1), 2) Incomplete POST Request Message Body (Attack-2), 3) Sending Connection Preface Only (Attack-3), 4) Incomplete GET/POST Request Header (Attack-4), and 5) Unacknowledged SETTINGS frame (Attack-5). It is not a good approach to mitigate these attacks by terminating the TCP connections having a connection lifetime greater than a predefined threshold. The reason is that incomplete requests are not uncommon on the Internet [7], [12], and such an approach may also terminate the connections from benign clients with poor TCP connection quality. Examples of such clients are remote clients with high latency and clients on low-grade cellular or satellite networks. Unlike traditional DDoS attacks, Slow HTTP/2 DoS attacks [12] need small computational power to execute on the attacker’s side. They involve sending very few requests to the web server to prevent it from serving benign clients [2], [11]. Since these attacks generate a smaller amount of traffic, they are stealthier and, thus, difficult to detect compared to traditional DDoS attacks [2], [11], [13].

2) Slow Rate DoS Attacks: HTTP/1.1 v/s HTTP/2: HTTP/2 was designed to fully utilize the TCP’s transmission capacity using built-in features such as server push, message multiplexing, flow control, etc. [12], [15]. These features require the transmission of data in different virtual HTTP/2 frames. One or more HTTP/2 frames are sent within a TCP segment simultaneously. However, in the case of HTTP/1.1, the web data is transmitted directly within the TCP segment without the concept of any virtual frames. Due to this significant difference in the operation of HTTP/1.1 and HTTP/2, the Slow Rate DoS attacks against HTTP/1.1 are ineffective against the HTTP/2 servers and vice versa. For example, launching Attack-4 against HTTP/2 and Slow Header attack against HTTP/1.1 require crafting incomplete requests and sending them to a web server. In the case of Attack-4, this is achieved by crafting a HEADERS frame with the END_HEADERS flag reset. On the other hand, in the case of Slow Header attack against HTTP/1.1, incomplete requests are crafted by removing the characters ‘\n\n\n’ from the web request.

3) Known Defence Mechanisms: Several works in the literature discuss defence strategies to counter attacks against HTTP/1.1. In [2], the authors presented a comprehensive state-of-the-art in this domain. Since the working of Slow Rate DoS attacks against HTTP/1.1 and HTTP/2 differ significantly, the detection schemes designed to detect Slow Rate DoS attacks against HTTP/1.1 are incompatible with HTTP/2 and, thus, can not detect Slow Rate DoS attacks against HTTP/2. As a result, we discuss only those works in this section that focus mainly on HTTP/2 security.

a) Detecting Slow HTTP/2 DoS attacks using Chi-square test: Tripathi and Hubballi [12] proposed a statistical abnormality measurement technique to detect Slow HTTP/2 DoS attacks. The approach declares a time interval \( \Delta T \) during the testing phase as normal or abnormal based on how much the HTTP/2 traffic profile generated during \( \Delta T \) deviates from the average HTTP/2 traffic profile generated during the training phase. The authors treated an HTTP/2 traffic profile \( P \) as an \( n \) dimensional vector with each component \( P_i \) of the vector \( P \) representing an incomplete request of type \( i \) received in a particular time interval. The comparison between the traffic profiles is made using the Chi-square statistic test. If the obtained \( \chi^2 \) value exceeds a predefined threshold, \( \Delta T \) is declared as an abnormal time interval. On the contrary, if the obtained \( \chi^2 \) value is less than the threshold, \( \Delta T \) is declared as a normal time interval. The experiments conducted by authors in [12] showed

| Responded  | 1st iteration (https:///<domain>) | 2nd iteration (https://www.<domain>) | 3rd iteration (http:///<domain>) | 4th iteration (http://www.<domain>) |
|-----------|----------------------------------|--------------------------------------|----------------------------------|----------------------------------|
| h2        | 335854                           | 017220                               | NA                               | NA                               |
| h2c       | NA                               | NA                                   | 002791                           | 000552                           |
| http/1.1  | 109461                           | 101441                               | 000000                           | 000000                           |
| Not responded | 054685                           | 045085                               | 144135                           | 143583                           |
| Total     | 500000                           | 164146                               | 146926                           | 144135                           |

\(^{1}\)Interested readers are requested to refer to our previous work [12] for a detailed working of these attacks.

**TABLE I HTTP Usage Statistics**
two limitations of this approach. First, it depends on the rate (number of incomplete requests per second) at which the attack is launched. Suppose the frequency of sending incomplete requests to launch the attack is low in a time interval. In that case, the detection performance of the scheme drops significantly, and it may declare that the interval contains only the normal traffic. Second, the detection performance of the scheme degrades with the decrease in the chosen time interval size. If the time interval size is reduced to achieve real-time detection behaviour, the detection performance degrades significantly. The detection performance of the approach presented in [12] is compared with our detection scheme in Section V-D.

b) Detecting multiplexed asymmetric HTTP/2 DDoS attack: In another work, Praseed and Thilagam [25] proposed an approach to detect the multiplexed asymmetric HTTP/2 DDoS attack. The proposed approach attempts to capture the legitimate browsing behaviour of users and, subsequently, detects abnormal requests if their behaviour deviates significantly from the normal one. The authors identified three defining characteristics of legitimate user behaviour. First, normal users access some pages and follow certain paths more often than others. On the other hand, the attackers typically do not possess this behaviour. Second, normal users take some time browsing the content of a webpage sent by the web server in response to a request. However, the same does not hold for an attacker. Third, the computational overhead generated by genuine web requests is uneven (high, medium, or low). However, the computational overhead due to attack web requests is typically high and medium. This mechanism can detect asymmetric HTTP/2 DDoS attacks with high accuracy, but it cannot detect Slow HTTP/2 DoS attacks because of the following three reasons:

- An attacker can send incomplete web requests for web pages in a specific order to mimic legitimate user behaviour.
- The second defining characteristic is not applicable in the case of Slow HTTP/2 DoS attacks, as the attacker would never receive a web page in response to an incomplete request.
- The computational overhead on the server due to incomplete web requests is always minimal as launching the Slow HTTP/2 DoS attacks does not require sending several multiplexed web requests into a single TCP connection.

As a result, the proposed detection scheme in [25] cannot detect Slow HTTP/2 DoS attacks.

Table II briefly summarizes how the work presented in this paper differs from the prior works on HTTP/2 security.

| References | Empirical Evaluation of Slow HTTP/2 DoS | Detection of Slow HTTP/2 DoS |
|------------|----------------------------------------|-----------------------------|
| Adi et al. [22] | ✗ | ✗ |
| Tripathi and Hubballi [12] | ✗ | ✓ (only offline) |
| Praseed and Thilagam [23] | ✗ | ✗ |
| Praseed and Thilagam [24] | ✗ | ✗ |
| Beckett and Sezer [16] | ✗ | ✗ |
| Beckett and Sezer [17] | ✗ | ✗ |
| Praseed and Thilagam [25] | ✗ | ✗ |
| Tripathi and Shaji [26] | ✓ | ✗ |
| This paper | ✓ | ✓ |

III. EMPIRICAL EVALUATION OF THE ATTACKS

We performed extensive experiments to test the behaviour of HTTP/2 supporting web servers on the Internet against Slow HTTP/2 DoS attacks. In this section, we first discuss the testbed created for conducting the required experiments. Subsequently, we discuss the behaviour of web servers against the attacks\(^2\).

\(^2\)The empirical evaluation results are also presented in our recent preliminary work [26].

A. Testbed Setup

We created a testbed setup for the empirical evaluation of Slow HTTP/2 DoS attacks, as shown in Fig. 1. Our testbed had four entities - 1) an attacker machine, 2) a transparent forward proxy, 3) an interceptor, and 4) the target web server.

**Attacker:** We designated a computer as the attacker machine and executed on it the attack scripts written in Python. These attack scripts sent the web servers exactly one incomplete HTTP/2 request of each attack (Attack-1 to 5) type. We sent only one incomplete request to observe the servers’ behaviour, and we did not intend to launch the attacks against the servers by sending multiple attack requests.

**Transparent Forward Proxy:** As discussed in the previous section, most web servers on the Internet support HTTP/2 over TLS (h2). Thus, to intercept and monitor otherwise encrypted HTTPS traffic, we configured a transparent forward proxy [27] in front of the attacker machine. This proxy connects to the web servers on behalf of the attacker. The proxy takes cleartext HTTP/2 requests from the attacker as input and establishes TLS connections\(^3\) with the web server of the domain present in the HOST field of the HTTP/2 requests. Subsequently, it forwards the encrypted web request to the server. On receiving the encrypted HTTP/2 response from the server, the proxy decrypts the response and forwards it to the attacker. It also re-plays the cleartext HTTP/2 traffic from the web server and the attacker to the interceptor, as shown in Fig. 1. The proxy uses the PCAP-OVER-IP method to

\(^3\)The TLS connections are established only with h2 servers. In the case of h2c servers, the proxy forwards the cleartext requests.

---

Fig. 1. Testbed setup.
We designated a computer as the interceptor that file. For each flow, we calculated the D pcap port, destination IP, and destination port. Servers. For each flow to compute the connection waiting time for different target web B. HTTP/2 Traffic Analysis

HTTP/2 supporting web servers we found on the Internet. This way, we intercepted the cleartext HTTP/2 traffic so that it could be analyzed later as described in Section III-B).

Interceptor: We designated a computer as the interceptor that intercepted the cleartext HTTP/2 traffic replayed by the proxy. The interceptor machine captured the inbound traffic to port 57012 using tcpdump and stored it in a pcap file.

Target Web Server: The target web server is one of the 356417 HTTP/2 supporting web servers we found on the Internet.

B. HTTP/2 Traffic Analysis

The captured HTTP/2 traffic at the interceptor was analyzed to compute the connection waiting time for different target web servers. For each flow4 in the pcap file, we calculated the time difference between the connection establishment (3-way handshake) and its termination (FIN-ACK packet exchange). This way, we obtained the time the web server waited before closing the TCP connection from which an incomplete request was sent. A CDF plot of the duration for which different web servers wait in case of each attack is shown in Fig. 2.

We can notice from the figure that approximately 3% HTTP/2 servers over the Internet waited for more than 360 seconds before closing a connection from which an attack-3 type web request was sent. On the other hand, approximately 50% servers waited for the same duration if an attack-1 type web request was sent. Similar other inferences can be drawn from the figure. Essentially, the figure shows that several HTTP/2 supporting web servers on the Internet are configured to wait for a significant time before closing a connection from which an incomplete request is sent. This is because a few clients over the Internet access the web resources using low-grade cellular or satellite networks. Since they possess poor TCP connection quality, their web requests are split into smaller chunks and transmitted to the web servers. To entertain such clients, the servers must be configured to wait unless they receive the complete web request. However, an attacker can exploit this behaviour of the web servers on the Internet and consume their connection queue space by sending multiple incomplete requests. Subsequently, the servers can not entertain legitimate requests sent by benign clients, which leads to a DoS scenario. These observations justify the requirement for an approach that can accurately detect Slow HTTP/2 DoS attacks in real-time. Taking motivation from this, we propose a detection approach in the next section to counter these attacks.

IV. PROPOSED DETECTION SCHEME

An HTTP/2 request involves exchanging different types of HTTP/2 frames. If we treat the exchange of these frames as unique events that occur one after the other, we can translate an ongoing interaction between a client and an HTTP/2 server into an event sequence. Since the order of the occurrence of HTTP/2 frames exchanged during the web requests can vary significantly, several distinct event sequences are possible. All possible normal event sequences can be stored in a database of characteristic normal patterns (observed sequences of exchanged HTTP/2 frames) and subsequently be used to detect anomalous sequences. An HTTP/2 request is considered an attack request if the event sequence generated corresponding to it does not exist in the normal database. However, simply checking the existence of an event sequence in the database to detect attack requests may result in several false positives. The reason is that there can be many possible unique event sequences (see Section V-A1), and maintaining a database of all the sequences to capture the complete normal behaviour of HTTP/2 would be a tedious task. An alternative approach to capture the normal behaviour of HTTP/2 is to maintain a database of all possible events only. However, in this case, the information regarding the distance between two events is lost, which is otherwise essential for detecting anomalies in the sequence.

To avoid these issues, we maintain a database of lookahead pairs extracted from the possible event sequences. Lookahead pairs are found to be useful in detecting the anomalies localised to one or more subsequences within the actual sequence [28]. Moreover, in Section V-A1, we show that the database of possible lookahead pairs can capture the normal behaviour of HTTP/2 with more completeness than the database of possible event sequences. Once we obtain the normal database of lookahead pairs during the learning phase, we assign a mismatch score to an event sequence during the detection phase based on the number of lookahead pairs extracted from it that do not exist in the normal database. The event sequence (and the corresponding web request) is declared anomalous only when its mismatch score exceeds a predefined threshold value.

A. Learning Phase

During the learning phase, our detection scheme builds a normal database of lookahead pairs extracted from the possible

---

4We differentiated the TCP flows using four parameters - source IP, source port, destination IP, and destination port.
TABLE III
INPUT AND OUTPUT VARIABLES USED IN ALGORITHMS 1 AND 2

| Variable | Description |
|----------|-------------|
| F        | Clear-text HTTP/2 flows |
| n        | Window size |
| Dlookahead | Lookahead pairs database |
| Ddelay   | Maximum delay database |
| t        | Mismatch score maximum threshold |
| f        | Incoming TCP flow |
| D        | Decision (Normal or anomalous sequence) |

Algorithm 1: Learning Phase.

Input: F, n
Output: Dlookahead, Ddelay
1: Dlookahead = {}; Ddelay = {};
2: for flow in F do
3:   seq = 'Start' → *';
4:   for frame in flow do
5:     e = frame.translateToEvent();
6:     seq+ = 'e' → *';
7:   end for
8:   seq+ = '→ End';
9:   l = extractLookaheadPairs(seq, n)
10:  Dlookahead.append(l)
11:  events = extractEvents(seq);
12:  i = 1;
13:  while i <= len(events) - 1 do
14:     delay = events[i].time() - events[i - 1].time();
15:     str = 'events[i]' → events[i]';
16:     if str not in Ddelay then Ddelay[str] = delay;
17:     else
18:         Ddelay[str] = max(delay, Ddelay[str]);
19:     end if
20:     i += 1;
21: end while
22: end for
23: return (Dlookahead, Ddelay);

event sequences and a database of the maximum normal delay possible between the occurrences of two consecutive events. Algorithm 1 describes the procedure for generating these databases during the learning phase. The input and output variables used in Algorithm 1 are described in Table III. Below we discuss the working of Algorithm 1 in detail.

1) Building the Lookahead Pairs Database (Dlookahead): We collect normal HTTP/2 traffic and construct flows by combining payloads of all the packets exchanged during the lifetime of a TCP connection. These flows are given as input along with the window size n to the learning phase. During this phase, one flow (say flow) is taken into account at a time (Step 2), and a sequence seq corresponding to flow is initialized to 'Start → *' (Step 3). This step represents the beginning of the flow, which is identified by the 3-way handshake. Subsequently, flow content is parsed using Deep Packet Inspection (DPI) to spot all HTTP/2 frames within it (Step 4). These frames are then translated into events (Step 5) based on the notations shown in Table IV and appended to seq (Step 6). Once all the events are appended to seq, ' → End' is also appended to seq (Step 8), representing the end of the flow (connection termination).

Consider a sample event sequence as follows:

\[
event1 → * → event2 → event3 → * → event4 → * → event5
\]

In this event sequence, the ' *' event corresponds to the end of an HTTP2 frame, while ' →' represents only a transition from one event to another.

Once an event sequence is generated, the lookahead pairs are extracted from it and stored in Dlookahead (Steps 9 - 10). For this, we slide a window of size n + 1 across the event sequence and capture which event follows which event within the sliding window. For example, we consider n = 3 and the sample event sequence. For each event, we capture the event that comes after it at spot 1, spot 2, and so on, up to spot n, as we slide the window across the event sequence. After sliding the window across the complete event sequence, we obtain the lookahead pairs, as shown in Table V.

A lookahead pair is stored in the database in the form of \(event[i]: event[j], k\) where \(k \leq n\) (window size) and corresponds to the distance between \(i^{th}\) and \(j^{th}\) events in a sequence. Once the database of lookahead pairs (Dlookahead) is built, our detection scheme refers to it to assign anomaly scores to the event...
sequences generated from incoming HTTP/2 requests during the detection phase.

2) Building the Maximum Delay Database (\(D_{delay}\)): Once the database of lookahead pairs (\(D_{lookahead}\)) is built, we build a database, \(D_{delay}\), of the maximum normal delay possible between the occurrences of two consecutive events, \(A\) and \(B\). This database is essential to detect the determined attackers who send the benign HTTP/2 frames to the server but with a prolonged delay between them to launch the Slow Rate DoS attacks. To build \(D_{delay}\), the events present in \(seq\) are first extracted and stored in a list \(events\) (Step 11). After this, \(i^{th}\) and \((i-1)^{th}\) events from the list \(events\) are taken into account at a time, and the delay between the timestamps of the occurrence of these two events (Step 14) is computed. This delay is stored in \(D_{delay}\) as a key : value pair where the key is \(\text{'events}[i-1] \rightarrow \text{events}[i]\)' (Step 15), and the value is delay. If this entry is not present in \(D_{delay}\), it will be stored in the database (Step 16). However, if the entry is present in \(D_{delay}\), the maximum between the current delay and the value already stored earlier in \(D_{delay}\) is chosen (Step 18).

B. Detection Phase

Algorithm 2 describes the procedure for detecting an anomalous event sequence during the detection phase. We use different threads for different purposes. The first thread executes the functions responsible for sniffing the incoming packets belonging to a particular flow \(f\), spotting frames in them, translating the spotted frames into corresponding events, and creating a sequence of the events. The second thread continuously checks timeouts between the events, and the third thread assigns anomaly scores to the sequences and updates them after the occurrence of each event. For brevity, we consider only one incoming TCP flow \(f\) and show how the event sequence generated from it is declared normal or anomalous. The input to Algorithm 2 is the databases \(D_{lookahead}\) and \(D_{delay}\), window size \(n\), mismatch score maximum threshold \(t\), and the incoming TCP flow \(f\) as described in Table III.

During the detection phase, the proposed scheme keeps on accumulating the packets belonging to flow \(f\), spotting HTTP/2 frames, translating them into corresponding events based on Table IV, and creating the event sequence \(seq\) corresponding to \(f\) (Steps 3 - 6). The \(seq\) is also continuously monitored to find the most recent event (\(last\_event\)) appended to it (Steps 8 - 9). Subsequently, the detection scheme refers to \(D_{delay}\) to retrieve the maximum delay (\(max\_delay\)) possible after the occurrence of \(last\_event\) (Step 10). If no new event is appended to \(seq\) within \(max\_delay\) duration, the detection scheme appends a ‘timeout string’ to \(seq\) (Step 11). The scheme keeps on appending a random timeout string to \(seq\) until a new event is appended to it. This ensures that if an attacker deliberately delays sending a request, the mismatch score of the sequence corresponding to that request increases quickly. As soon as the length of \(seq\) exceeds the window size \(n\), the detection scheme starts extracting the lookahead pairs from \(seq\) (Step 15). For each extracted pair, the scheme subsequently checks if the lookahead pair exists in the lookahead pairs database \(D_{lookahead}\). If the pair does not exist in \(D_{lookahead}\), the number of mismatches for \(seq\) is increased by one (Step 17). Once all the lookahead pairs extracted from \(seq\) are checked against the entries in \(D_{lookahead}\), a mismatch score is assigned to \(seq\) (Step 20) (see Section IV-B1). If the mismatch score exceeds the predefined threshold \(t\), \(seq\) is considered anomalous (Step 21). However, if the \(seq\) is complete (i.e., the TCP connection is terminated) and its mismatch score is still less than \(t\), \(seq\) is considered normal (Steps 22 - 23).

| Index-1 | Index-2 | Index-3 |
|---------|---------|---------|
| event1 | event2 | event3 |
| *      | event2 | event4 |
|        | event4 | event5 |
|        | event3 | *      |
|        | event4 | *      |
| event4 | *      | event5 |

**Algorithm 2:** Detection Phase.

**Input:** \(D_{lookahead}, D_{delay}, n, t, f\)

**Output:** \(D\)

1: \(seq = \text{""};\)
2: while True do
3:     Continuously sniff packets belonging to \(f\);  
4:     for frame in \(f\) do
5:         \(e = \text{frame.translateToEvent}();\)
6:         \(seq = e \rightarrow \text{seq}\);  
7:     end for
8:     events = seq.extractEvents();  
9:     last_event = events[\([-1]\)]; 
10:    \(\text{max}\_delay = \text{max}(D_{delay}[\text{'last\_event \rightarrow E'}])\)  
11:    if \((\text{current}\_\text{time}() - \text{last\_event.time}()) \geq \text{max}\_delay\) then  
12:        seq = seq + \(\text{'\rightarrow TO} \rightarrow \text{seq'}\) \(\text{where}\)  
13:            \(i = 1, 2, \ldots\);  
14:    end if
15:    if mismatch = 0;  
16:    if len(events) > \(n\) then  
17:        l = seq.extractLookaheadPairs();  
18:    for pair in \(\text{pair}\) do  
19:        if \(\text{pair not in } D_{lookahead}\) then mismatch += 1;  
20:    end if
21:    if mismatch > \(t\) then \(seq\) is anomalous;  
22:    else if mismatch < \(t\) and events[\([-1]\)] = \(\text{End}\) then  
23:        seq is normal;  
24:    end if
25:    end if
26: end while
TRIPATHI: DELAYS HAVE DANGEROUS ENDS: SLOW HTTP/2 DOS ATTACKS INTO THE WILD AND THEIR REAL-TIME DETECTION

### TABLE VI

| Event 1 | Event 2 | Event 3 |
|---------|---------|---------|
| event1  | event2  | event2  |
| event2  | event3  | event4  |
| event3  | event4  | event5  |

Please refer to the colour version for better illustration.

1) Assigning Mismatch Scores: Consider an example where $D_{\text{Lookahead}}$ contains the entries as shown in Table V, and we generate the following event sequence from an HTTP/2 request at a particular time instance $t_i$:

$$\text{event}1 \rightarrow \text{event}2 \rightarrow \text{event}3 \rightarrow \text{event}4 \rightarrow \text{event}5$$

For $n=3$, the lookahead pairs extracted from this event sequence are shown in Table VI.

Since the lookahead pairs shown in red in Table VI do not exist in Table V, these are counted as mismatches. Subsequently, the mismatch score of the event sequence is computed as the ratio of the mismatch counts to the total possible mismatch counts. For instance, if we consider an event sequence of length $L$ and a lookahead (window size) of $n$, the maximum pairwise mismatch count can be calculated as follows:

$$n(L - n) + (n - 1) + (n - 2) + \cdots + 1 = n(L - (n + 1)/2)$$

Thus, in the example discussed earlier, the mismatch score of the sequence is $7/21 (= 0.33)$. The mismatch score of the event sequence is computed after every new event appended to the event sequence. It will be continued unless either of the following two conditions is met:

1. The mismatch score of the sequence exceeds the predefined threshold. In this case, the corresponding HTTP/2 request from which the sequence is being generated will be treated as an attack request.
2. The $\text{End}$ event is appended to the event sequence. After appending the $\text{End}$ event, if the mismatch score of the sequence is less than the predefined threshold, the corresponding HTTP/2 request will be treated as benign. However, if the score exceeds the threshold, the request will be treated as an attack request.

### V. EXPERIMENTS

#### A. Learning Phase

Researchers in the security community expended considerable time and effort to generate datasets for various research purposes. Table VII shows a list of such datasets and their description.

The first seven datasets in this table were published earlier than the RFC 7540 [15] (published in 2016) that describes HTTP/2. Thus, they do not contain HTTP/2 traces. On the other hand, the ISOT cloud intrusion detection dataset published in 2018 contains the traces generated from HTTP/1.1 (not HTTP/2) flood DoS attacks, among other application layer attacks. Since these datasets do not contain clear-text HTTP/2 traffic (also confirmed by authors in [25]), we created a testbed similar to the one shown in Fig. 3 to capture clear-text normal HTTP/2 traffic during the learning phase. Subsequently, we used it to build the lookahead pairs database and the maximum delay database.

**Please refer to Section III-A for the details of the Interceptor.**
attempt the assignments online for their academic grading. From the web server, we collected approximately 5 GB of HTTP/2 traffic. During the COVID-19 pandemic, the students visited the website from different geographical locations, and thus, the collected HTTP/2 traffic exhibited the property of actual Internet behaviour.

**User:** The users in our testbed were the students of our institute who accessed the web server to complete their assignments. They were asked to use the latest HTTP/2 supporting web browsers. We also practised a few non-technical formalities to ensure that the students do not send malicious traffic to poison the collected traffic.

**Reverse TLS Proxy:** We configured a reverse TLS proxy application called PolarProxy \[27\] in front of the web server. This proxy was owned by the website administration team and was responsible for the tasks shown in Fig. 3.

1) **Capturing the Normal Behaviour of HTTP/2:** It is essential to decide what size of the database should be built to capture the normal behaviour of HTTP/2. For this purpose, we collected 14094 HTTP/2 flows, and for each flow, we generated an event sequence. It resulted in 7221 unique event sequences. We also extracted the lookahead pairs from the unique event sequences and obtained 227, 377, 471, 621, and 737 unique lookahead pairs for window size \(n=3, 4, 5, 6, \text{ and } 7\), respectively. The plots in Figs. 4(a) and (b) show how the number of unique event sequences and lookahead pairs increases as we increase the number of HTTP/2 flows. We can notice from Fig. 4(a) that the number of new event sequences kept increasing with the increase in the number of flows. Thus, it is not easy to estimate how much HTTP/2 traffic should be collected beforehand to capture the complete normal behaviour of HTTP/2.

On the contrary, as can be noticed from Fig. 4(b), virtually no new lookahead pairs were witnessed beyond a point \(p\) (i.e., approximately 9400 flows). Thus, we could assume that the database of lookahead pairs captured the virtually complete normal behaviour of HTTP2 at \(p\). From this study, we can also establish that a database of lookahead pairs could capture the expected behaviour of HTTP/2 with more completeness than a database of unique event sequences.

**B. Detection Phase**

During the detection phase, we extended the setup shown in Fig. 3 by including one more entity and designating it as the attacker. The attacker machine was connected to the Internet and was responsible for sending the web server a small number of attack (Attack-1 to 5) requests at regular intervals. Moreover, the attacker terminated a connection\(^6\) from which an attack request was sent as soon as 100 seconds progressed after the connection establishment. This ensured such attack requests did not deplete the server’s connection queue space. Closing these connections deliberately after 100 seconds did not alter the event sequences generated from the attack requests because our detection scheme could classify such requests as an attack or benign way before the aforementioned 100 seconds (see Section V-C). The benign requests were also generated during the detection phase as the students attempted their assignments.

The reverse TLS proxy in our testbed captured the benign and attack HTTP/2 requests generated by the students and the attacker, respectively, and forwarded their clear-text versions to the Interceptor. Our detection scheme on the Interceptor generated event sequences from the requests and subsequently classified them as normal or anomalous depending on their mismatch scores. We conducted this experiment in five different time intervals such that the window size \(n\) was fixed, but the mismatch score threshold \(t\) was varied across the intervals. In this way, we could analyze the effect of varying \(t\) on the detection performance of the scheme. We used four metrics to evaluate the detection performance - **Accuracy**, **False Positive Rate** (FPR), **Recall**, and **Precision**. The detection performance of the scheme in terms of these four metrics is shown in Table VIII. We can notice from the table that **Accuracy** and **Precision** slightly increased as we increased \(t\) to 0.02 from 0.01. It was

\(^6\)A connection was terminated using FIN-ACK TCP packet.
because of the reduced number of FP cases (a normal sequence incorrectly classified as anomalous). However, as we further increased t, Accuracy decreased because of more FN cases (an anomalous sequence incorrectly classified as normal). Moreover, Recall and Accuracy decreased drastically as we increased t to 0.03 from 0.02. It was because the anomalous sequences with a mismatch score in the range [0.02, 0.03] were considered normal, resulting in many FN cases. Due to these misclassifications, we obtained lower detection accuracy. The same holds as we further increased the t value.

1) Mismatch Scores of Normal and Anomalous Event Sequences: Fig. 5 shows a CDF plot of the mismatch scores of normal and anomalous sequences for n=5. It is clear from the figure that the mismatch scores of virtually all normal/benign sequences are distributed within the range [0, 0.01]. Thus, if the mismatch score threshold t is chosen in the range [0, 0.01], it will lead to several false positives. Also, it can be noticed from Fig. 5 that the number of anomalous sequences with a mismatch score in the range [0.02, 0.03] remains the same. Thus, to minimize the number of FP without any increase in FN, the mismatch score threshold t can be chosen in the range [0.01, 0.02]. It can also be noticed from Fig. 5 that the number of anomalous sequences with a mismatch score ≥0.02 starts decreasing with the increase in the mismatch score. Thus, if t is chosen ≥0.02, the anomalous sequences with a mismatch score <0.02 would be considered normal sequences, leading to an increase in FN count. This will result in the degradation of the detection performance of the proposed scheme. As a result, choosing t in the range [0.01, 0.02] would result in the best detection performance of the scheme, as also apparent from Table VIII.

2) Detection Performance vs Different Values of n: To analyze the effect of varying the window size n on the detection performance of the scheme, we experimented with four more time intervals of 8 hours each. The mismatch score threshold t was the same in all four intervals, but the value of n was varied across these intervals, as shown in Table IX. To check the detection performance for a value of n, the lookahead pairs database generated for the same value of n was given as input to the detection scheme. The detection scheme subsequently refers to this database to count mismatching lookahead pairs and assign mismatch scores to the event sequences (Steps 16 - 20 of Algorithm 2). The obtained results are shown in Table IX. We can notice from the table that varying the window size does not virtually affect the proposed scheme’s detection accuracy.

C. Detection Performance Evaluation Parameters

Lookahead pair extraction time: While observing the effect of varying n on the detection performance of the scheme, we noticed that the time required to extract the lookahead pairs from an event sequence increases if we increase n. This observation is shown in Fig. 6. This results in the delayed classification of an event sequence as normal or anomalous. Thus, choosing a smaller window size is essential to calculate the mismatch scores and make the appropriate decision as soon as possible.

Detection latency: It is essential for an active detection scheme to detect anomalies as soon as possible so that an alarm can be raised and necessary preventive measures can be taken. Thus, we measured how fast our proposed detection scheme could detect anomalous event sequences. Fig. 7 shows the result of this measurement for n=5. We can notice from the figure that the detection scheme could detect virtually all anomalous sequences in less than 30 seconds.
Computational overhead: To estimate the computational overhead caused by executing the detection scheme at the Interceptor, we monitored for 3 hours the RAM and the CPU usage using the Python library psutil. The Interceptor computer had an AMD Ryzen 7 5800H processor and 16 GB (15.3 GB usable) RAM. In the first half of this observation period, we monitored the overhead without executing the detection scheme (i.e., when the computer was idle). We executed the detection scheme in the second half and monitored the overhead. The computational overhead in terms of CPU usage is shown in Fig. 8. It can be noticed from the figure that the overhead on the CPU due to the execution of the detection scheme was marginally greater than the overhead on the CPU when the Interpreter computer was idle. Moreover, we also observed that the RAM usage was approximately 10% ($\approx 1.60$ GB) when the Interpreter computer was idle. On executing the detection scheme, the RAM usage increased for approximately the first three minutes and then remained virtually constant at 12.5% ($\approx 1.92$ GB). Due to the small computational overhead, the proposed detection scheme can thus be an ideal solution for detecting Slow HTTP/2 DoS attacks in real-time.

D. Detection Performance Comparison

As discussed earlier in Section II-B3, there is only one defence approach [12] known in the literature to detect Slow HTTP/2 DoS attacks. However, popular ML clustering algorithms can also be adapted to detect attacks. Thus, in this section, we test the detection performance of 1) the previously known technique [12], and 2) two ML clustering algorithms (DBSCAN and K-means) to detect Slow HTTP/2 DoS attacks.

1) Detection Performance of [12]: Authors in [12] proposed an offline defence approach to detect Slow HTTP/2 DoS attacks. We tested the detection performance of this approach on the HTTP/2 traces collected at Interceptor so that its performance can be compared with our detection scheme’s performance. We split the HTTP/2 traces into two equal parts and used the first and second parts for training and testing, respectively. To test the effect of varying time interval sizes on the detection performance of this scheme, we chose different time interval sizes, as shown in Table X.

In the first iteration, we split the training period into smaller time intervals of 5 minutes and then created a mean HTTP/2 normal traffic profile for the entire training period. We also split the testing period into smaller time intervals of 5 minutes and then created an HTTP/2 traffic profile for each interval.
Subsequently, we compared the traffic profiles created for different time intervals of the testing period with the mean traffic profile created for the entire training period. This comparison was made using $\chi^2$ statistical test at a significance level of 0.05. If the calculated $\chi^2$ value for a time interval was greater than the threshold, the interval was reported as containing attack traffic. However, if an interval’s calculated $\chi^2$ value was less than the threshold, the interval was reported as containing normal traffic. In the subsequent iterations, we varied the time interval sizes and recorded the detection performance, as shown in Table X. It can be noticed from the table that the accuracy of the approach degrades significantly as we reduce the time interval duration to achieve real-time detection. This was because the Recall of the approach drops sharply as we decrease the time interval size. This was also evident in the experimental results presented by the authors [12]. Thus, the approach presented in [12] is not ideal for real-time detection of Slow HTTP/2 DoS attacks, unlike our scheme, which could accurately detect attacks in real-time (as shown earlier in Fig. 7). Moreover, as demonstrated by the authors in [12], the performance of the detection approach depends on the attack rate. Thus, if the attack rate is very low, it is difficult for the approach to detect anomalous time intervals. On the other hand, the detection performance of our scheme does not depend on the rate at which Slow HTTP/2 DoS attacks are launched.

2) Detection Performance of ML Clustering Algorithms: We tested the detection performance of two ML clustering algorithms - DBSCAN and K-means. For this purpose, we used the normal and the attack event sequences generated from our testbed so that the performance of both algorithms could be compared with our detection scheme’s performance. In the first step, the subsequences of a fixed size $x$ are extracted from the normal event sequences and fed to the DBSCAN algorithm. This generated a set of clusters $(n_1)$ into which the subsequences were classified. In the second step, the subsequences of size $x$ are extracted from both normal and attack event sequences and fed to the DBSCAN algorithm. This also resulted in a set of clusters $(n_2 \subseteq n_1)$ into which the subsequences were classified. Subsequently, the number of mismatched subsequences for each sequence was counted. A normal subsequence was said to be mismatched if classified to a cluster $c_1 \subseteq n_1$. Likewise, an attack subsequence was said to be mismatched if it is classified to a cluster $c_2 \subseteq n_1$. After this, a mismatch score for each event sequence was calculated. The mismatch score for an event sequence $e$ was calculated as the ratio of the count of mismatching subsequences extracted from $e$ to the total number of subsequences of size $x$ that could be extracted from $e$. If the mismatch score for $e$ exceeded a predefined threshold, it was treated as an attack event sequence. However, $e$ was treated as a normal event sequence if the score was less than the threshold. The same experiment was also performed by considering K-means clustering. The best detection performance that could be achieved for both algorithms is shown in Table XI for four different window sizes. In this table, column ‘K’ refers to the number of predefined clusters for the K-means clustering. We can notice from the table that the proposed approach could achieve better performance than the ML clustering algorithms in detecting Slow HTTP/2 DoS attacks.

### Table XI: Detection Performance of ML Clustering Algorithms

| Algorithm  | K window size | Accuracy | Recall | FPR | Precision |
|------------|---------------|----------|--------|-----|-----------|
| DBSCAN     | -             | 69.81    | 0.00   | 0.00| Undefined |
|            | -             | 77.08    | 0.00   | 0.00| 100.00   |
|            | -             | 80.77    | 0.00   | 0.00| 100.00   |
|            | -             | 83.97    | 0.01   | 0.00| 099.92   |
| K-means    | 2             | 70.09    | 0.00   | 0.00| 100.00   |
|            | 3             | 59.75    | 0.00   | 0.00| 100.00   |
|            | 4             | 50.76    | 0.00   | 0.00| 100.00   |
|            | 2             | 60.14    | 0.00   | 0.00| 100.00   |
|            | 3             | 61.16    | 0.00   | 0.00| 100.00   |
|            | 4             | 70.99    | 0.00   | 0.00| 100.00   |

VI. CONCLUSION

Slow Rate DoS attacks are a matter of grave concern for server administrators because of two prominent reasons. First, these attacks require significantly less computational power and, thus, can be launched even from mobile devices. Second, these attacks generate minimal traffic, due to which they are highly stealthy. The effect of these attacks on HTTP/2 servers over the Internet had not been analyzed earlier. Moreover, the known mechanisms to counter the attacks against HTTP/1.1 can not counter them against HTTP/2. Thus, in this work, we attempted to bridge these gaps by first performing an empirical evaluation of Slow HTTP/2 DoS attacks on the Internet and subsequently proposing a scheme to detect these attacks in real-time. Our experiments showed that several HTTP/2 servers on the Internet delay closing the connections from which incomplete requests are sent. This behaviour makes the Internet web servers vulnerable to Slow HTTP/2 DoS attacks. Our proposed scheme to detect these attacks treats an HTTP/2 interaction as a sequence of events and checks for anomalies in this sequence. Our experiments showed that the proposed scheme could accurately detect the attacks in real-time with a marginal computational overhead. Also, the approach could outperform the ML clustering algorithms and the previously known methods to counter Slow HTTP/2 DoS attacks.

ACKNOWLEDGMENTS

The author would like to thank the Netressec team for supporting this research with a free L1 license of PolarProxy.

REFERENCES

[1] A. Praseed and P. S. Thilagam, “DDoS attacks at the application layer: Challenges and research perspectives for safeguarding web applications,” IEEE Commun. Surv. Tut., vol. 21, no. 1, pp. 661–685, First Quarter 2019.
[2] N. Tripathi and N. Hubballi, “Application layer denial-of-service attacks and defense mechanisms: A survey,” ACM Comput. Surv., vol. 54, no. 4, pp. 1–33, 2021.
[3] U. Inayat, M. F. Zia, S. Mahmood, H. M. Khalid, and M. Benbouzid, “Learning-based methods for cyber attacks detection in IoT systems: A survey on methods, analysis, and future prospects,” Electronics, vol. 11, no. 9, 2022, Art. no. 1502.
[4] S. Ashraf, M. H. Shawon, H. M. Khalid, and S. Muyeen, “Denial-of-service attack on IEC 61850-based substation automation system: A crucial cyber threat towards smart substation pathways,” Sensors, vol. 21, no. 19, 2021, Art. no. 6415.
[5] Cyberphysical Infrastructures in Power Systems: Architectures and Vulnerabilities, M. S. Mahmoud, H. M. Khalid, and M. M. Hamdan, Eds., Cambridge, MA, USA: Academic Press, 2022, pp. 389–400.

[6] H. M. Khalid and J. C.-H. Peng, “A bayesian algorithm to enhance the resilience of WAMS applications against cyber attacks,” IEEE Trans. Smart Grid, vol. 7, no. 4, pp. 2026–2037, Jul. 2016.

[7] N. Tripathi, N. Hubballi, and Y. Singh, “How secure are web servers? An empirical study of slow HTTP DoS attacks and detection,” in Proc. Int. Conf. Availability Rel. Secur., 2016, pp. 454–463.

[8] N. Tripathi and N. Hubballi, “A probabilistic anomaly detection scheme to detect DHCプ starvation attacks,” in Proc. Int. Conf. Adv. Netw. Telecommun. Syst., 2016, pp. 1–6.

[9] N. Tripathi and N. Hubballi, “Preventing Time Synchronization in NTP Broadcast Mode,” Comput. Secur., vol. 102, 2021, Art. no. 102135.

[10] Imperva,”HTTP/2: In-depth analysis of the top four flaws of the next generation web protocol," 2016. [Online]. Available: https://www.imperva.com/docs/Imperva_HII_HTTP2.pdf

[11] E. Cambiaso, M. Aiello, M. Mongelli, and I. Vaccari, “Detection and classification of slow DoS attacks targeting network servers,” in Proc. 15th Int. Conf. Availability Rel. Secur., 2020, Art. no. 61.

[12] N. Tripathi and N. Hubballi, “Slow rate denial of service attacks against HTTP/2 and detection,” Comput. Secur., vol. 72, pp. 255–272, 2018.

[13] S. T. Zargar, J. Joshi, and D. Tipper, “A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks,” IEEE Commun. Surv. Tut., vol. 15, no. 4, pp. 2046–2069, Fourth Quarter 2013.

[14] SANS ISC InfoSec Forums,”Slowloris and iranian DDoS attacks,” 2009. [Online]. Available: https://isc.sans.edu/forums/diary/Slowloris+and+Iranian+DDoS+attacks/6622/

[15] M. Belshe, R. Peon, and M. Thomson, “Hypertext transfer protocol version 2 (HTTP/2),” 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7540

[16] D. Beckett and S. Sezer, “HTTP/2 cannon: Experimental analysis on HTTP/1 and HTTP/2 request flood DDoS attacks,” in Proc. Int. Conf. Emerg. Secur. Technol., 2017, pp. 108–113.

[17] D. Beckett and S. Sezer, “HTTP/2 tsunami: Investigating HTTP/2 proxy amplification DDoS attacks,” in Proc. Int. Conf. Emerg. Secur. Technol., 2017, pp. 128–133.

[18] Core, 2019. [Online]. Available: https://httpd.apache.org/docs/2.4/mod/core.html

[19] mod_antiloris, 2013. [Online]. Available: https://sourceforge.net/projects/mod-antiloris/

[20] mod_limitipconn, 2002. [Online]. Available: http://dominia.org/djao/limitipconn.html

[21] mod_retimeout, 2019. [Online]. Available: https://httpd.apache.org/docs/trunk/mod/mod_retimeout.html

[22] E. Adi, Z. A. Baig, P. Hingston, and C.-P. Lam, “Distributed denial-of-service attacks against HTTP/2 services,” Cluster Comput., vol. 19, no. 1, pp. 79–86, 2016.

[23] A. Praseed and P. S. Thilagam, “Modelling behavioural dynamics for asymmetric application layer DDoS detection,” IEEE Trans. Inf. Forensics Secur., vol. 16, pp. 617–626, 2021.

[24] A. Praseed and P. S. Thilagam, “Multiplexed asymmetric attacks: Next-generation DDoS on HTTP/2 servers,” IEEE Trans. Inf. Forensics Secur., vol. 15, pp. 1790–1800, 2019.

[25] A. Praseed and P. S. Thilagam, “Fuzzy request set modelling for detecting multiplexed asymmetric DDoS attacks on HTTP/2 servers,” Expert Syst. Appl., vol. 186, pp. 115–697, 2021.

[26] N. Tripathi and A. K. Shaji, “Defer no time, delays have dangerous ends: Slow HTTP/2 DoS attacks into the wild,” in Proc. IEEE 14th Int. Conf. Commun. Syst. Netw., 2022, pp. 194–198.

[27] NETRESEC,”“PolarProxy,” 2020. [Online]. Available: https://www.netresec.com/pages/PolarProxy

[28] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection for discrete sequences: A survey,” IEEE Trans. Knowl. Data Eng., vol. 24, no. 5, pp. 823–839, May 2012.

[29] “KDD Cup 1999 Data,” 1999. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[30] CAIDA The UCSD,”“DDoS attack2007,” Dataset, 2007. [Online]. Available: https://www.caida.org/catalog/datasets/ddos-20070804_dataset/

[31] “Anonymized internet traces 2008,” 2008. [Online]. Available: https://catalog.caida.org/details/dataset/passive_2008_pcap

[32] M. Tavaallea, E. Bagheri, W. Lu, and A. A. Ghorbani,”“A detailed analysis of the KDD CUP’99 data set,” in Proc. IEEE Symp. Comput. Intellic. Secur. Defense Appl., 2009, pp. 1–6.

[33] S. Saad et al., “Detecting P2P botnets through network behavior analysis and machine learning,” in Proc. 9th Annu. Int. Conf. Privacy Secur. Trust, 2011, pp. 174–180.

[34] A. Shiravi, H. Shiravi, M. Tavaallea, and A. A. Ghorbani,”“Toward developing a systematic approach to generate benchmark datasets for intrusion detection,” Comput. Secur., vol. 31, no. 3, p. 357–374, 2012.

[35] N. Moustafa and J. Slay,”“UNSW-NB15: A comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set),” in Proc. Mil. Commun. Inf. Syst. Conf., 2015, pp. 1–6.

[36] A. Aldibri, I. Traore, and B. Moa,”“Data sources and datasets for cloud intrusion detection modeling and evaluation,” in Cloud Computing for Optimization: Foundations, Applications, and Challenges. Berlin, Germany: Springer, 2018, pp. 333–366.

[37] A. Aldibri, I. Traore, B. Moa, and O. Nwamu,”“Hypervisor-based cloud intrusion detection through online multivariate statistical change tracking,” Comput. Secur., vol. 88, 2020, Art. no. 101646.