Slow Rate Denial of Service Attacks Against HTTP/2 and Detection

Article *in* Computers & Security · September 2017

DOI: 10.1016/j.cose.2017.09.009

2 authors, including:

Nikhil Tripathi
Indian Institute of Technology Indore
15 PUBLICATIONS **116** CITATIONS

SEE PROFILE

Slow Rate Denial of Service Attacks Against HTTP/2 and Detection

Nikhil Tripathi, Neminath Hubballi*

***Abstract*—HTTP/2 is a newly standardized protocol designed to efficiently utilize the TCP's transmission rate and has other advantages compared to HTTP/1.1. However its threat vectors are not completely understood yet. Our contribution in this paper is threefold. First we describe few new threat vectors of HTTP/2 which are Slow Rate DoS attacks and can be launched by injecting specially crafted HTTP requests. We perform an empirical evaluation of these attacks against popular web servers and report that majority of web servers are vulnerable to these attacks. We also test the effectiveness of proposed attacks using both clear text and encrypted HTTP/2 requests and find that the attack is effective independent of the request type. Second we compare structurally similar attacks with HTTP/1.1 and report that HTTP/2 has more threat vectors compared to its predecessor. Third we propose an anomaly detection scheme which uses chi-square (χ^2) test between traffic profiles generated in normal and attack scenarios to detect these attacks.**

***Index Terms*—HTTP/2, HTTP/1.1, DoS Attacks, Vulnerability Assessment, Chi-square Test, Anomaly Detection.**

I. Introduction

With the development of highly efficient network infrastructure, research in networking community is now focusing towards the development of robust application layer protocols which can utilize the potential and capability of underlying infrastructure to the fullest. For example, application layer Hypertext Transfer Protocol (HTTP/1.1) [19] uses TCP as transport layer protocol for reliable data transmission. It is argued that HTTP/1.1 uses TCP inefficiently by not transmitting at the full rate [7]. This can be viewed as a negative impact on application performance. Problems like Head-of-Line (HoL) blocking (a HTTP request resulting into voluminous response size which blocks other small requests) restrict HTTP/1.1 to efficiently use TCP services. This motivated researchers for the development of HTTP/2 [7], the recent version of HTTP protocol. HTTP/2 not only supports all the basic features of HTTP/1.1 but it is also very efficient in utilizing TCP transmission capability. As HTTP/2 is a new protocol, the research community has not yet paid much attention on the security issues or vulnerabilities in HTTP/2. There are only few works available in literature which discussed some security flaws in HTTP/2 [24], [2], [22]. We believe that a proper understanding of possible threats that may affect normal operation of a wildly popular protocol such as HTTP/2 is essential in order to develop versatile detection and mitigation methods.

The authors are with Discipline of Computer Science and Engineering, Indian Institute of Technology Indore, India. (Email:{phd1401101002,neminath}@iiti.ac.in).
*Corresponding Author

In this paper, we present few new threat vectors of HTTP/2 which we found after a careful analysis of working of the protocol (HTTP/2). These threat vectors are Slow Rate DoS attacks which can thwart a HTTP/2 server from serving legitimate clients. Subsequently we propose a statistical abnormality measurement technique that uses chi-square statistic test to detect any deviation in the HTTP/2 traffic profile created with normal HTTP/2 traffic. This method can identify deviations in network traffic patterns due to presence of anomalous connections originating from these Slow Rate attacks. In particular, we make following contributions in this paper:

- We propose novel Slow Rate HTTP/2 DoS attacks which are effective against majority of popular web servers. To the best of our knowledge, proposed attacks are the first reported Slow Rate DoS attacks against HTTP/2 protocol.
- We perform the vulnerability assessment of both HTTP/1.1 and HTTP/2 protocols and show that the latter has relatively more number of threat vectors that can be exploited to launch Slow Rate DoS attacks.
- We perform experiments using both clear text and encrypted HTTP/2 requests and find that the attack is effective in both the cases.
- We also propose a novel detection mechanism to detect these Slow Rate DoS attacks. We test it in a real network setup and show that it can detect anomalies in the HTTP/2 traffic with very high accuracy.

Rest of the paper is organised as follows. We provide an overview of different DoS/DDoS attacks and HTTP/2 basics in Section II. In Section III, we propose Slow Rate HTTP/2 DoS attacks and discuss the behaviour of different web servers against these attacks. A comparison of vulnerability assessment of HTTP/1.1 and HTTP/2 protocols is presented in Section IV. We describe proposed detection method and its experimental evaluation in Section V and Section VI respectively. Other related work is discussed in Section VII followed by conclusion in Section VIII.

II. Background

In this section, we describe few well known DoS/DDoS attacks and their limitations in terms of bandwidth requirement and stealthiness. We also discuss few basics of HTTP/2 protocol which we consider is important in order to understand the attacks proposed in this paper.

A. DoS/DDoS Attacks

In this subsection, we briefly mention some of the transport- and application-level DoS/DDoS attacks and highlight their limitations from malicious client's perspective.

1) Transport-level DDoS Flooding Attacks: Transport-level DoS attacks are mostly launched using TCP, UDP packets. Malicious client either disrupts the victim's connectivity by exhausting its network bandwidth (e.g. spoofed/non-spoofed UDP flood [33]) or exploits implementation bugs of transport layer protocols in order to consume excess amount of victim's resources (e.g. TCP SYN flood, ACK & PSH ACK flood, etc. [33]). Also, malicious client may use reflection and amplification to launch attacks such as ICMP Echo Request Flood attack and Smurf attack [17]. Another class of DoS attacks called as Low-Rate TCP targeted DoS attacks [26] exploits TCP's retransmission timeout mechanism where a malicious client provokes a TCP flow to repeatedly enter a retransmission timeout state by sending high-rate but short-duration bursts. These DDoS attacks exploit the network and trasport layer operations, thus require a lot of malicious client's bandwidth to flood the victim as compared to Slow Rate application layer DoS attacks. Also, transport-level DDoS attacks can be detected very easily because of the excessive amount of traffic generated by these attacks [44], [27].

2) Application-level DDoS Flooding Attacks: These attacks also target to disrupt victim client by exhausting its resources however they require less bandwidth and thus, are stealthier as compared to transport-level DDoS attacks[44]. To launch these attacks, malicious client either uses reflection (e.g. VoIP flooding [6], [20]) and amplification technique (e.g. DNS amplification attack [17]) or simply protocol specific request flooding (e.g. HTTP flooding attacks [8], DHCP Starvation Attack [23], [36] and etc.). These attacks involve sending of complete requests at a very high rate in order to overwhelm the victim client.

3) Application-specific Slow Rate DoS Attacks: Application-specific Slow Rate DoS attacks (e.g. Slowloris HTTP/1.1 and FTP DoS attacks [44]) require very small number of incomplete requests to create DoS scenario. To launch the attack, malicious client establishes multiple connections with the victim and sends an incomplete request from each of these connections. Since incomplete requests belonging to these connections interact very slowly with server, the server stores them in a connection queue space till these requests are completely served. Once all the available space in this queue is occupied, no legitimate connections are entertained by server, thereby, causing DoS attack. These DoS attacks require minimal bandwidth and a highend server can easily be taken down even using minimal computational resources. These attacks are highly stealth as they generate very small amount of traffic and mimic normal traffic behaviour. Slow Rate HTTP/2 DoS attacks proposed in this paper belong to this category only and they also target the number of free connection slots available in web server's connection pool.

4) Application Layer Protocol Independent Slow Rate DoS Attacks: Few works in the literature discussed variants of Slow Rate DoS attacks which are independent of application layer protocol being targeted. SlowReq [5], Slowcomm [10] and Slow Next [12] attacks belong to this category. Along with HTTP, these attacks can create DoS scenario against protocols such as FTP and SMTP as well. Though in this paper, we discuss these attacks considering HTTP protocol only. SlowReq and Slowcomm attacks possess some similarities with Slowloris attack as all these attacks involve sending of incomplete and pending requests. However, these two attacks also have the ability to detect connection closes in reasonable times and re-establish such connections as soon as they are closed. This makes SlowReq and Slowcomm much more effective as compared to simple Slowloris attack. Slow Next, on the other hand, involves sending of valid and legitimate requests to server. This attack makes use of persistent connection feature of HTTP/1.1 protocol to maintain the established connection in open state even after receiving server's response for a particular request sent earlier. Before expiry of waiting time for a connection to be in open state, another legitimate valid request is sent on the same established connection in order to reset the timer. Once enough number of such connections are established with the server, it becomes unavailable for genuine users. Due to presence of complete requests, server parses it successfully and sends back the response lawfully. As a result, this attack is stealthier as compared to previously mentioned Slow Rate DoS attacks. Since these attacks are protocol independent, they are also termed as $meta$ attacks [11].

B. HTTP/2

HTTP/2 protocol is defined in RFC 7540 [7] which was standardized recently in May, 2015. The operation of HTTP/2 is significantly different from HTTP/1.1 though the semantics of the original protocol remain the same. HTTP/2 improves the ability of an application to efficiently utilize the transmission bandwidth which its predecessor is lacking. Limitations like only one outstanding request per connection at a time makes HTTP/1.1 inefficient considering the capacity of modern internet infrastructure. To overcome these limitations, HTTP/2 provides features like message multiplexing, prediction of resource requirement at client-side in advance and compression of header information into serialized header blocks. These new control features have significantly improved the communication speed at which clients and web servers can interact. With the help of message multiplexing, each connection can have multiple concurrent streams which allow several requests that can be outstanding at a time in a connection. Streams are bidirectional sequence of frames exchanged between client and server within an HTTP/2 connection. Due to multiple streams multiplexed over a single connection, a HTTP client can send multiple concurrent HTTP requests, thereby, maximizing bandwidth utilization. Streams also resolve the issue of Head-of-Line (HoL) blocking which is a major limitation in HTTP/1.1. Every stream is uniquely identified by an unsigned 31 bit integer. Also, HTTP/2 messages are divided into different types of independent binary frames and each type serves different purpose. We discuss few types of frames below.

- Connection Preface: Connection Preface (contains a magic string $PRI * HTTP/2.0\backslash r\backslash n\backslash r\backslash nSM \backslash r\backslash n\backslash r\backslash n$) is used to establish the initial settings for a HTTP/2 connection and final confirmation of the protocol in use.

- HEADERS and CONTINUATION Frames: HEADERS frame is used to carry a header block. In case a header block is large enough not to fit in a single HEADERS frame, CONTINUATION frames are used to transmit remaining parts of header block.
- DATA Frame: This frame is used to carry message body sent by the endpoints. For example, client uses DATA frame to carry message body of a POST request while server uses DATA frame to carry response to a client's GET request.
- SETTINGS Frame: This frame is used by endpoints for the purpose of negotiating connection parameters like initial window size, maximum concurrent streams, etc.
- WINDOW_UPDATE Frame: WINDOW_UPDATE frame is used to indicate the number of bytes that the sender is willing to accept by its peer in addition to the existing HTTP flow-control window.
- GOAWAY Frame: This frame is used either to tear off a established connection between endpoints or to indicate some serious error condition. RFC 7540 defines various error codes that GOAWAY frame carries to convey the reason behind connection or stream error. Some of the error codes are:
  - NO_ERROR: This code indicates graceful shutdown of a connection.
  - PROTOCOL_ERROR: This code indicates an unspecific protocol error. This code is usually transmitted when there is no more specific code available.
  - INTERNAL_ERROR: This code is transmitted in case an endpoint encounters an unexpected break in the normal working of protocol.
  - FLOW_CONTROL_ERROR: An endpoint transmits this code if peer violates the flow control negotiations made.
  - SETTINGS_TIMEOUT: In case an endpoint does not receive acknowledgement of a SETTINGS frame sent by it, it sends a GOAWAY frame with SETTINGS_TIMEOUT code.
  - FRAME_SIZE_ERROR: An endpoint transmits this code if it receives a frame with an invalid size.

Figure 1 shows exchange of frames which are involved in the normal operation of HTTP/2 protocol. The sequence is as follows:

- 1^{st} Payload from Client to Server: As soon as TCP connection is established, client sends Connection Preface, SETTINGS frame and WINDOW_UPDATE frame on stream having identification number 0 while HEADERS frame is sent on another stream multiplexed over same connection. Both these streams are part of one HTTP/2 payload.
- 2^{nd} Payload from Server to Client: As soon as server receives the previous HTTP/2 payload, it acknowledges the receipt of SETTINGS frame by sending an empty SETTINGS frame on stream having identification number 0. Along with this frame, server also sends WINDOW_UPDATE frame and a non-empty SETTINGS frame. As a response to the GET request, server also sends HEADERS and DATA frame(s) on another stream back to the client.
- 3^{rd} Payload from Client to Server: Once client receives the 2^{nd} HTTP/2 payload, it acknowledges the SETTINGS frame sent by the server and finally closes the connection successfully by transmitting a GOAWAY frame.

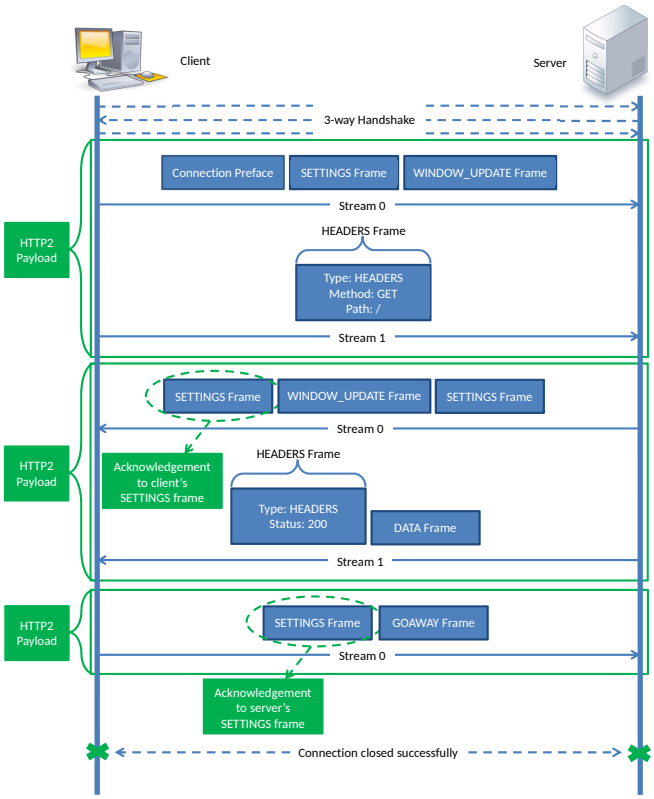

Fig. 1: HTTP/2 Working

III. Proposed Slow Rate HTTP/2 DoS Attacks

In this section, we present new threat vectors of HTTP/2 which are Slow Rate DoS attacks and can be launched by injecting specially crafted HTTP requests. Slow Rate DoS attacks [11], [44], mentioned in the literature, target the number of free connection slots available on a server. Similarly, attacks proposed in this paper also focus on depleting all the free connection slots available in web server's connection pool. In order to consume all the free slots, malicious client establishes enough number of connections and sends specially crafted requests from each connection. Since these special requests hold back established connections for a long duration, server is unavailable to entertain requests sent by genuine HTTP clients for that amount of time. This results into DoS scenario. In next subsection, we discuss the testbed setup used to evaluate the effectiveness of proposed attacks to create a DoS scenario.

A. Testbed Setup

We tested proposed attacks on four popular [35] HTTP/2 supporting web servers namely Apache 2.4.23, Nginx 1.10.1, H2O 2.0.4 and Nghttp2 1.14.0. All four servers were tested in

their default configurations and with the latest versions available at the time of writing. All these servers were configured on a computer having 4 GB of physical memory and quad core processor. This computer was running Kali 2.0 operating system. We hosted a sample website in each server's home directory. This website contains a homepage in which fifty images are embedded. We also designated two other computers in the network as malicious and genuine clients. Each of these clients were having 4 GB of physical memory and dual core processor and running Ubuntu 16.04 LTS operating system. Malicious client was used to launch the proposed attacks while the genuine client was used to check the server's availability during attack scenario. Due to unavailability of any full-fledged tool that can provide flexibility to modify some parameters of HTTP/2 frames while testing the protocol, we implemented these attacks in python. We used h2load [21] benchmarking tool to create and send legitimate requests from genuine client to check the server's availability during attack. In next few subsections, we describe how different Slow Rate DoS attacks can be launched and the effect of these attacks on different web servers.

B. Attack-1: Slow Rate DoS Attack using Complete GET Header and Zero SETTINGS_INITIAL_WINDOW _SIZE

To generate the attack, malicious client sends a HTTP/2 payload having a SETTINGS frame with SETTINGS_INITIAL_WINDOW_SIZE field set to zero and a complete GET request as shown in Figure 2. SETTINGS_INITIAL_WINDOW_SIZE field indicates the sender's currently available capacity for receiving the data in bytes from its peer. On receiving this payload, server assumes that client can not receive any data right now. Thus, server waits to receive WINDOW_UPDATE frames from malicious client. Malicious client, on the other hand, never sends WINDOW_UPDATE frames to server which makes server wait for a particular time duration depending on the web server's software implementation.

An attack, called Slow Read attack, using somewhat similar strategy was tested by Imperva [24] against HTTP/2. Similar to Attack-1, Slow Read attack also targets the number of connections a web server can handle at a time. To launch this attack, authors set very small SETTINGS_INITIAL_WINDOW_SIZE and send GET requests from several concurrent streams multiplexed over a single TCP connection. Servers like Apache 2.4.17 and Apache 2.4.18 are vulnerable to the attack because these software versions dedicate one thread per stream which results into consumption of all worker threads once enough number of streams are created. However, with the release of latest security patch [14], Apache 2.4.20 and later server versions are not vulnerable any longer as these versions dedicate threads on per connection basis instead of per stream basis. Thus, instead of creating multiple streams over a single TCP connection, Attack-1 requires malicious client to establish multiple connections with each connection having only one stream[1].

[1]This also reduces load on the malicious client.

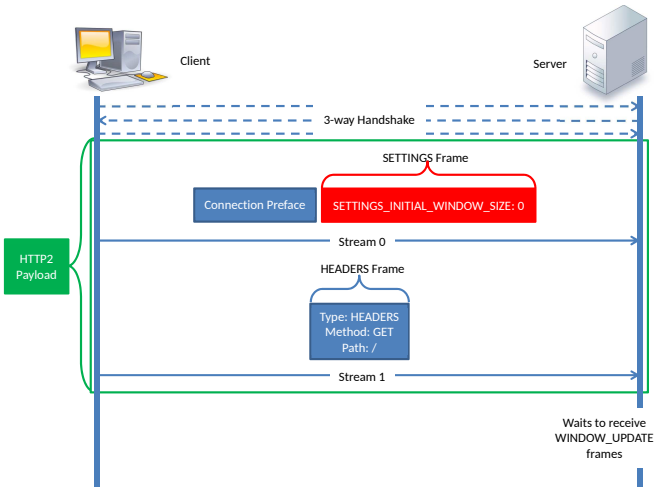

Fig. 2: Attack-1. SETTINGS frame with SETTINGS_INITIAL_WINDOW_SIZE set to 0.

TABLE I: Connection Waiting Time at Server if Attack-1 Payload is sent

Server	Waiting Time (in seconds)
Apache	300
Nginx	60
H2O	Indefinite
Nghttp2	60

Behaviour of Web Servers Against Attack-1: In case of Attack-1, Apache waits for 300 seconds before closing the connection. After this waiting time, server closes the connection by sending a GOAWAY frame with NO_ERROR code back to malicious client. Both Nginx and Nghttp2 wait for 60 seconds before closing the connection. After this waiting time, Nginx and Nghttp2 close the connection by sending a RST_STREAM frame with PROTOCOL_ERROR and INTERNAL_ERROR code respectively. H2O is also vulnerable to this attack as it waits for indefinite time to receive WINDOW_UPDATE frame. Table I shows amount of time for which different server implementations wait before closing the connection if the payload corresponding to Attack-1 is sent.

C. Attack-2: Slow Rate DoS Attack using Complete POST Header

There are four types of flags in HEADERS frame. Two among these flags are END_STREAM and END_HEADERS. When reset, END_STREAM flag indicates that the sender is still having DATA frames to be transmitted on the stream. END_HEADERS flag, when set, indicates that the HEADERS frame contains an entire header block and this frame will not be followed by any CONTINUATION frame. To launch the attack, malicious client sets and resets the END_HEADERS and END_STREAM flags of the HEADERS frame respectively and sends a complete POST request contained within this frame as shown in Figure 3. As soon as server receives this HEADERS frame, it assumes that though it has received

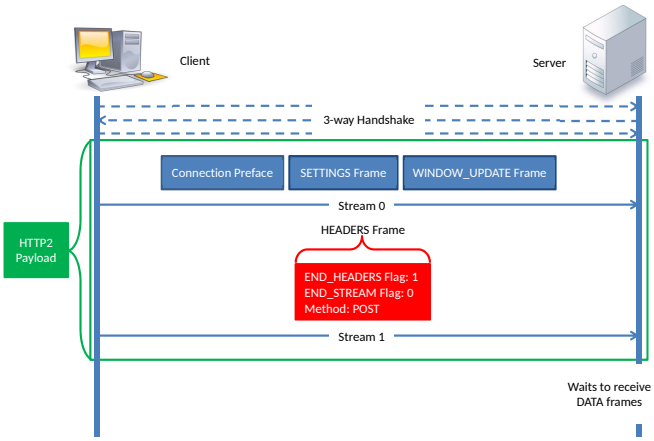

Fig. 3: Attack-2. HEADERS frame with END_HEADERS set and END_STREAM reset

TABLE II: Connection Waiting Time at Server if Attack-2 Payload is sent

Server	Minimum Waiting Time (in seconds)	Maximum Waiting Time (in seconds)
Apache	600	Indefinite
Nginx	30	Indefinite
H2O	10	Indefinite
Nghttp2	10	975

complete POST request header (due to END_HEADERS flag set), one or more DATA frames are yet to be received (due to END_STREAM flag reset). Depending on the server's software implementation, it waits for a particular amount of time to receive DATA frames before closing the connection.

Behaviour of Web Servers Against Attack-2: If malicious client does not send any DATA frame after sending POST request to server, Apache, Nginx and H2O close the connection after 600, 30 and 10 seconds respectively. After this timeout, Apache and H2O send back GOAWAY frame with NO_ERROR code while Nginx sends back GOAWAY frame with PROTOCOL_ERROR code to close the connection. However, if malicious client keeps sending DATA frames having END_STREAM flag reset at regular intervals, all these three servers wait for an indefinite amount of time by resetting their expiry timers. Nghttp2 closes the connection after 10 seconds by sending GOAWAY frame with SETTINGS_TIMEOUT code if no DATA frame is received. In case malicious client tries to extend this timeout duration by sending a DATA frame, then also Nghttp2 closes the connection after 10 seconds by sending GOAWAY frame with SETTINGS_TIMEOUT code. However, if malicious client acknowledges the SETTINGS frames which was sent by the server after connection establishment, server waits for malicious client to close the connection. If malicious client does not close the connection, Nghttp2 waits for a time period of 975 seconds. After this timeout, Nghttp2 closes the connection without sending any frame back to the client. Table II shows minimum and maximum time for which different server implementations wait before closing the connection if payloads corresponding to Attack-2 are sent.

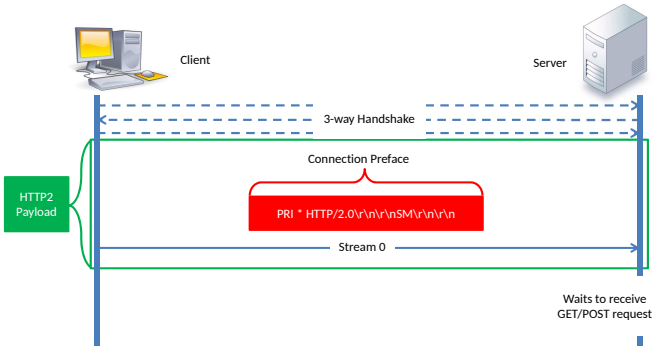

Fig. 4: Attack-3. First HTTP/2 payload with only Connection Preface

D. Attack-3: Slow Rate DoS Attack using Connection Preface

After connection establishment, malicious client sends Connection Preface to the server as shown in Figure 4. Server honours this message and starts waiting to receive a GET/POST HTTP request. Malicious client, on the other hand, never sends any HTTP request at all. This, in turn, forces server to wait for a particular amount of time to receive the HTTP request. This allows malicious client to get enough waiting time at server to create DoS scenario.

Behaviour of Web Servers Against Attack-3: Once Connection Preface without any GET/POST request is received by the Apache web server, it waits for 300 seconds in the hope of receiving GET/POST request from the malicious client before closing the connection. After this timeout, server closes the connection by sending GOAWAY frame with NO_ERROR code. In case malicious client tries to reset this time counter by sending the Connection Preface again, server immediately closes the connection by sending GOAWAY frame with PROTOCOL_ERROR code. Thus, malicious client can not extend this timeout period. Nevertheless, 300 seconds is still a long duration for which the server maintains a connection. Nginx waits for 30 seconds in the hope of receiving GET/POST request. After this timer expires, server closes the connection by sending GOAWAY frame with PROTOCOL_ERROR code. However, if malicious client keeps sending Connection Preface at regular intervals, Nginx waits for an indefinite amount of time. H2O is not vulnerable to this attack as it closes the connection just after 10 seconds by sending GOAWAY frame with NO_ERROR code. In case malicious client tries to extend this timeout duration by again sending Connection Preface, H2O immediately closes the connection by sending GOAWAY frame with FRAME_SIZE_ERROR code. Nghttp2 also closes the connection after 10 seconds by sending GOAWAY frame with SETTINGS_TIMEOUT code. However, similar to attack discussed in Section III-C, if malicious client acknowledges the SETTINGS frames sent by the server after connection establishment, it waits for 975 seconds in the hope that malicious client will close the connection. After this timeout, Nghttp2 closes the connection without sending any frame back to client. Table III shows minimum and maximum time for which different server implementations wait before closing the connection if payloads corresponding to Attack-3 are sent.

TABLE III: Connection Waiting Time at Server if Attack-3 Payload is sent

Server	Minimum Waiting Time (in seconds)	Maximum Waiting Time (in seconds)
Apache	300	300
Nginx	30	Indefinite
H2O	10	10
Nghttp2	10	975

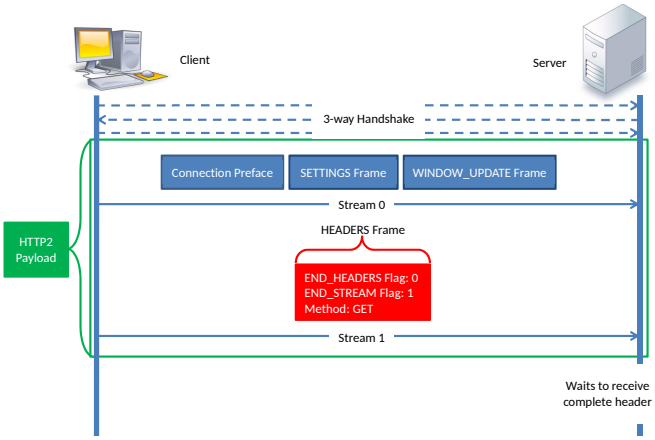

Fig. 5: Attack-4 using GET Request. HEADERS frame with END_HEADERS reset and END_STREAM set

E. Attack-4: Slow Rate DoS Attack using Incomplete GET/POST Header

Malicious client can launch this attack using either GET or POST request. To launch the attack using GET request, malicious client sends a HEADERS frame with END_HEADERS and END_STREAM flag reset and set respectively as shown in Figure 5. END_HEADERS flag, when reset, indicates that the HEADERS frame must be followed by one or more CONTINUATION frames. The last CONTINUATION frame must have END_HEADERS flag set. When server receives such HEADERS frame, server assumes that it has received incomplete header and it should wait to receive complete header block. To launch the attack using POST request, malicious client sends a HEADERS frame containing POST request and having both END_STREAM and END_HEADERS flag reset. Thus, there are two flavours of Attack-4 depending on which request type HEADERS frame is containing.

Behaviour of Web Servers Against Attack-4: Before closing a connection, Apache and Nginx wait for 300 and 90 seconds respectively after receiving a HEADERS frame having END_STREAM flag set and END_HEADERS flag reset. However, if malicious client keeps sending CONTINUATION frames having END_HEADERS flag reset at regular intervals, Apache waits for an indefinite amount of time by resetting its expiry timer again and again. However, Nginx do not reset the expiry timer and after the waiting time expires, Nginx closes the connection by sending GOAWAY frame with PROTOCOL_ERROR code. Both H2O and Nghttp2 just wait for 10 seconds after receiving such HEADERS frame. After timeout, H2O and Nghttp2 close the connection by sending GOAWAY frame with NO_ERROR and HEADER_TIMEOUT code respectively. However, if malicious client keeps sending CONTINUATION frames having END_HEADERS flag reset at regular intervals, H2O waits for an indefinite amount of time while Nghttp2 waits for 60 seconds. After this timeout, Nghttp2 closes the connection by sending RST_STREAM frame with INTERNAL_ERROR code. Table IV shows minimum and maximum time for which different server implementations wait before closing the connection if payloads corresponding to Attack-4 are sent.

TABLE IV: Connection Waiting Time at Server if Attack-4 Payload is sent

Server	Minimum Waiting Time (in seconds)	Maximum Waiting Time (in seconds)
Apache	300	Indefinite
Nginx	90	90
H2O	10	Indefinite
Nghttp2	10	60

F. Attack-5: Slow Rate DoS Attack using SETTINGS frame

According to RFC 7540, a SETTINGS frame sent by one endpoint must be acknowledged by another endpoint. This acknowledgement is sent in the form of a SETTINGS frame of zero length. If the sender of SETTINGS frame does not receive acknowledgement within a reasonable amount of time, it may close the connection by issuing a connection error of type SETTINGS_TIMEOUT [7]. To create DoS using this technique, malicious client sends legitimate complete GET or POST request. Figure 6 shows example of this attack using legitimate GET request. Server responds to this request by sending DATA frame along with two SETTINGS frames. First SETTINGS frame is used to acknowledge SETTINGS frames sent by the client just after the connection establishment while second SETTINGS frame includes parameters that server wants to negotiate with the client. The second SETTINGS frame must be acknowledged by malicious client. However, malicious client does not acknowledge this SETTINGS frame sent by server. As a result, vulnerable web servers wait for a particular time to receive the acknowledgement. Malicious client can choose not to acknowledge SETTINGS frame of server while sending a POST request also. Thus, there are two flavours of this attack similar to Attack-4.

We also observed that few server implementations, even after receiving the acknowledgement for SETTINGS frame by client, wait for the client to close the connection. If client does not close the connection, these web servers wait for reasonably large amount of time before closing the connection themselves. Thus, even acknowledging SETTINGS frame also allows malicious client to create DoS for few server implementations.

Behaviour of Web Servers Against Attack-5: Once Apache and H2O send the response back for malicious client's request, they close the connection after 5 and 10 seconds respectively by sending back GOAWAY frame with NO_ERROR code without waiting for their SETTINGS frames to be acknowledged. As a result, both these servers are not vulnerable

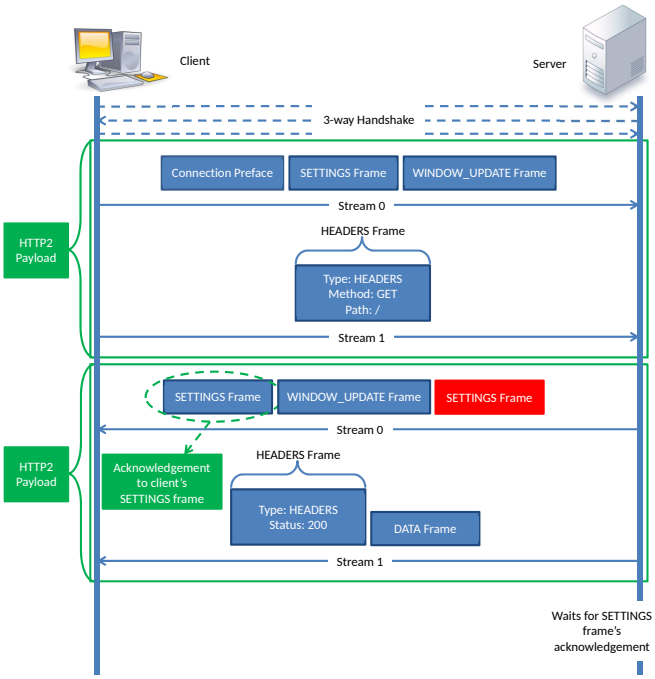

Fig. 6: Attack-5 using GET Request. Malicious client never acknowledges SETTINGS frame sent by server

TABLE V: Connection Waiting Time at Server if Attack-5 Payload is sent

Server	Minimum Waiting Time (in seconds)	Maximum Waiting Time (in seconds)
Apache	5	5
Nginx	180	180
H2O	10	10
Nghttp2	10	975

to this attack. However, Nginx waits for 180 seconds to receive the acknowledgement before closing the connection. After this timeout, it closes the connection by sending GOAWAY frame with NO_ERROR code. Nghttp2 waits for 10 seconds after serving a request sent by the malicious client. After this timeout, server closes the connection by sending GOAWAY frame with SETTINGS_TIMEOUT code. However, if the SETTINGS frame sent by server after connection establishment are acknowledged, server waits for 975 seconds in the hope that malicious client will close the connection. After this timeout, Nghttp2 closes the connection without sending any frame back to client. Table V shows minimum and maximum time for which different server implementations wait before closing the connection if payloads corresponding to Attack-5 are sent.

Table VI shows number of connections different web servers can handle at a time. Malicious client needs to establish these number of connections (within their respective timeout periods) in order to create a DoS scenario using either of the proposed attacks.

TABLE VI: Number of Simultaneous Connections Servers can handle

Server	Number of Connections
Apache	150
Nginx	2060
H2O	1024
Nghttp2	1142

G. Proposed Attacks over TLS

In order to test the effectiveness of proposed attacks using encrypted HTTP/2 requests, we repeated the above experiments by configuring web servers to use TLSv1.2 [15] while serving the requests. We used OpenSSL [30] to generate the required digital certificate and private key. All the four servers were configured to use the generated certificate and private key. For the purpose of packet crafting and automated TLS handshake, we implemented a python script on the malicious client. This python script used scapy-ssl_tls python library [34]. The script first performed the TLS handshake for negotiation purpose and then encrypted the specially crafted HTTP/2 request using the session key exchanged for the session. We hardcoded the required HTTP/2 payload in *hex* format in the script. The resulting encrypted request was then transmitted to server. The python script was automatically executed multiple times using a shell script so that malicious client could establish multiple concurrent connections with the server and send an encrypted HTTP/2 request from each of those connections. Once enough number of connections were established, genuine client was used to check the availability of server. From this experiment, we observed that malicious client required same number of connections to create DoS scenario using either clear text or encrypted HTTP/2 requests. The number of connections required to cause DoS against each of the four web servers is shown in Table VI. Moreover, the amount of time durations for which different proposed attacks (as shown in Tables I-V) hold the connection queue space was same for clear text and encrypted request. Thus, the proposed attacks are found to cause similar effects on each server irrespective of clear text or encrypted HTTP/2 requests.

IV. Comparison of Slow Rate DoS Attacks in HTTP/1.1 and HTTP/2

In this section, we compare the two HTTP versions (HTTP/1.1 and HTTP/2) based on the similarity of threat vectors. For this comparison, we chose Apache 2.4.23 web server as a base. Though the attacks proposed in this paper are specific to HTTP/2 but three of them are structurally similar to known Slow Rate DoS attacks of HTTP/1.1 as follows.

A. Slow Read Attack and Attack-1

Slow Read attack [32] against HTTP/1.1 server requires a malicious client to send a legitimate GET request after 3-way handshake and then immediately advertises a receiver window size of 0 byte. As a result, server stops sending data although it holds the connection in the hope of receiving a

non-zero window size advertisement from the client. Client on the other hand, never advertises non-zero window size, thus, forcing server to wait for indefinite time. Similar to Slow Read attack, malicious client sends a SETTINGS frame with SETTINGS_INITIAL_WINDOW_SIZE parameter set to 0 to launch Attack-1. However, Apache HTTP/1.1 server is not vulnerable to Slow Read attack as it immediately closes the TCP connection which advertises a receiver window size of 0 bytes. On the other hand, Attack-1 is effective against Apache HTTP/2 server.

B. Slow Message Body Attack and Attack-2

In order to launch Slow Message Body [37] attack against HTTP/1.1 server, malicious client sends complete POST request but message body is sent in smaller chunks so as to keep server waiting to receive complete message body. Usually, malicious client sends HTTP POST requests with a large *Content-Length* value as compared to the actual size of message body. On receiving this request, web server assumes that only a part of the message is received and rest of the message body is still pending. As a result, server keeps its resources busy waiting for the rest of message. To create DoS, malicious client establishes at least 150 connections and sends such POST requests from each connection. HTTP/1.1 server waits for an indefinite amount of time in the hope of receiving complete message body. This finally leads to DoS. Attack-2 also involves sending of smaller message chunks at regular intervals to maintain the connection in open state. Both Slow Message Body attack and Attack-2 are effective to create DoS scenario in HTTP/1.1 and HTTP/2 respectively.

C. Slow Header Attack and Attack-4

To launch Slow Header attack [44], also known as Slowloris attack, malicious client sends incomplete HTTP GET requests by not transmitting the string "$\backslash r \backslash n \backslash r \backslash n$" which denotes the end of a HTTP header. Instead, it sends bogus header fields at regular intervals to keep the connection alive. The malicious client establishes several connections and sends such incomplete GET requests from each connection to create DoS. Attack-4 against HTTP/2 server also involves sending of incomplete GET request header and then subsequent bogus headers to keep the connection alive. Slow Header attack can create DoS scenario if malicious client establishes at least 150 connections with the server. Before closing the connection, server waits for 300 seconds if no subsequent header parts are sent. However, this waiting time increases to 990 seconds if subsequent header parts are sent at regular intervals. SlowReq and Slowcomm attacks also cause similar effect on Apache server. Similarly, HTTP/2 is also vulnerable to Attack-4 as discussed in Section III-E.

Table VII summarizes behaviour of both the protocols in presence of different types of Slow Rate DoS attacks. We can notice that HTTP/1.1 is vulnerable to two types of Slow Rate DoS attacks while HTTP/2 is vulnerable to four types of Slow Rate DoS attacks (For Attack-3 and Attack-5, there are no comparable attacks in HTTP/1.1) which are discussed in this paper. Thus, it is evident that HTTP/2 has relatively more number of threat vectors in comparison to Slow Rate DoS attacks of HTTP/1.1.

D. Comparison of Attacks in HTTP/1.1 and HTTP/2 with Encrypted Requests

We performed a comparison between threat vectors in HTTP/1.1 and HTTP/2 by considering encrypted requests also using libraries mentioned in Section III-G. As discussed earlier, proposed attacks behaved similar against HTTP/2 irrespective of clear text or encrypted requests. In case of HTTP/1.1 protocol also, we observed that effectiveness of Slow Rate DoS attacks remains same independent of the type of request as the timeout period and number of connections required to create DoS scenario do not depend on the clear text or encrypted requests. Thus, the comparison summary shown in Table VII holds valid for clear-text and encrypted HTTP requests as well.

V. Detecting Proposed Attacks

In this section, we propose an anomaly detection scheme to detect Slow Rate HTTP/2 DoS attacks. Our detection scheme has two phases of operation as training phase and testing phase. In training phase, we create a normal HTTP/2 profile by collecting legitimate web traffic coming to and going from a server over a period of x observation intervals. In testing phase, detection system compares the current traffic profile with the profile generated during training phase. For this comparison, we propose a distance measurement technique using chi-square test statistics. In next three subsections we describe candidate features that are used to detect proposed attacks, basics of chi-square test statistic and how to adapt it for detection purpose respectively.

A. Feature Selection

In this subsection, we discuss a set of features which our detection technique uses to detect the proposed attacks. In particular, these features are number of such events which occur if any of the proposed attacks are present. For example, an event corresponding to a value of 0 in SETTINGS_INITIAL_WINDOW_SIZE field of a SETTINGS frame occurs due to presence of Attack-1. Similarly, few other features are also defined to detect presence of other types of attacks as shown in Table VIII. Feature-1 corresponds to number of SETTINGS frame having SETTINGS_INITIAL_WINDOW_SIZE set to 0. Similarly, Feature-2 and Feature-4 correspond to number of HEADERS frame with either END_STREAM_FLAG or END_HEADERS_FLAG reset respectively. Feature-3 corresponds to number of flows[2] having Connection Preface only while Feature-5 corresponds to number of server's SETTINGS frames which are not acknowledged. Due to unavailability of any stable HTTP/2 parsing library, we wrote a program using jNetPcap [25] Java library to extract the required features.

[2]Each HTTP/2 flow or connection is uniquely identified by the combination of source IP address and source port number.

TABLE VII: Comparison of Slow Rate Attacks of HTTP/1.1 and HTTP/2

HTTP/1.1			HTTP/2		
Attack Type	**Number of Connections Required to Create DoS**	**Maximum Waiting Time at Server**	**Attack Type**	**Number of Connections Required to Create DoS**	**Maximum Waiting Time at Server**
Slow Read Attack	Not Vulnerable	0	Attack-1	150	300
Slow Message Body Attack	150	Indefinite Time	Attack-2	150	Indefinite Time
-	-	-	Attack-3	150	300
Slow Header/SlowReq/ Slowcomm Attacks	150	990	Attack-4	150	Indefinite Time
-	-	-	Attack-5	Not vulnerable (Ngnix and Nghttp2 web servers are vulnerable)	5

TABLE VIII: Candidate Features

Features	Description	Type
Feature-1	SETTINGS frame having SETTINGS_INITIAL_WINDOW_SIZE set to 0	Numeric
Feature-2	HEADERS frame having END_STREAM_FLAG Reset	Numeric
Feature-3	Flows having Connection Preface only	Numeric
Feature-4	HEADERS frame having END_HEADERS_FLAG Reset	Numeric
Feature-5	Server's SETTINGS frames which are not acknowledged	Numeric

B. Chi-Square Test

Chi-square test [16] is a statistical hypothesis test used to determine if there is a significant deviation in the observed and expected frequencies of one or more categories. This test requires proposing two hypotheses- $null$ and $alternate$. Null hypothesis states that there is no significant difference between the expected and observed frequency while alternate hypothesis states that observed frequencies are significantly deviated from the expected ones due to some reason. The confidence with which we can conclude whether this difference is not created only due to chance is known as *significance level*, denoted by α. Usually a significance level of 0.05 [13] is considered for most of the scientific experiments. The chi-square test statistic is defined as shown in Equation 1

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{1}$$

where O_i and E_i are the observed and expected number of cases in i^{th} category respectively and n is the number of categories. χ^2 value is very small in case O_i and E_i for each category are closer to each other. Higher the difference between O_i and E_i, higher will be the χ^2 value.

1) Null and Alternate Hypotheses: The null and alternate hypotheses for a chi-square test can be stated as shown in Equation 2

$$H_N{:}\, O_i = E_i; H_A{:}\, O_i \neq E_i \tag{2}$$

In case the null hypothesis is found to be true, O_i and E_i for each category are closer to each other. This results into smaller value of numerator in Equation 1, thereby, causing smaller χ^2 value. However, if O_i is dissimilar to what has been expected in null hypothesis, $O_i - E_i$ value in Equation 1 is large. In this case, we obtain a larger χ^2 value and thus, alternate hypothesis is accepted. The significance level (α) and chi-square distribution table can be used to determine the threshold χ^2 value. If the obtained χ^2 value exceeds this threshold, we can reject null hypothesis H_N. However, if χ^2 value is below the predefined threshold, we do not have enough evidence to reject H_N and accept alternate hypothesis H_A.

C. Adapting Chi-Square Statistic for Slow Rate HTTP/2 DoS Attack Detection

Chi-square statistical test can easily be adapted to detect proposed attacks. For detection purpose, our null and alternate hypotheses are H_N : *Time interval* ΔT *does not contain anomalous flows* and H_A : *Time interval* ΔT *contains anomalous flows* respectively. Considering Equation 1 for detection purpose, n is the number of features, O_i corresponds to Feature-i received in a particular time interval of testing phase and E_i corresponds to mean of Feature-i received in different time intervals of training phase.

1) Normal HTTP/2 Profile Creation: In order to make our detection system learn normal HTTP/2 behaviour, we collect legitimate HTTP/2 traffic coming to and going from the web server over a period of x time intervals each of duration ΔT. The HTTP/2 profile, E, is then created by taking mean of each feature received in different time intervals of training phase. E is a n dimensional vector with each component E_i of the vector representing Feature-i. E_i is estimated using Equation 3

$$E_i = \frac{\sum_{t=1}^{x} e_{it}}{x} \tag{3}$$

where e_{it} corresponds to Feature-i received in t^{th} time interval of training phase and x is the total number of time intervals in the training phase.

2) Comparing Profile Generated during Training and Testing Phase: Once the detection system is trained with the normal HTTP/2 profile, we use it to detect presence of attacks from $(x+1)^{th}$ interval. We generate a profile, O, after every

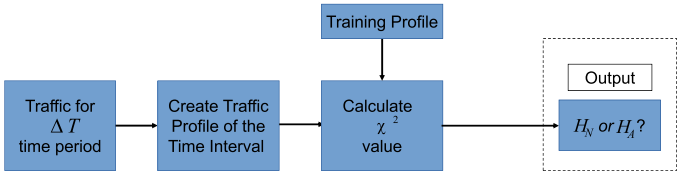

Fig. 7: Detector Working

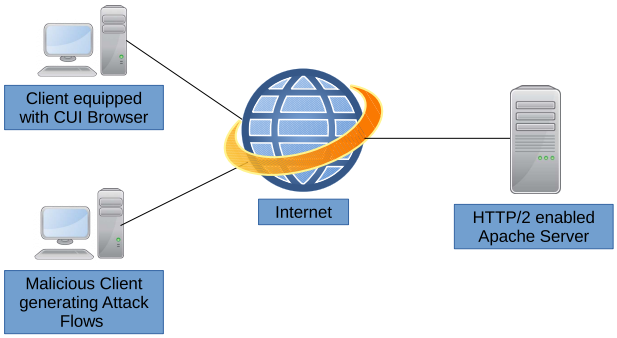

Fig. 8: Testbed Architecture

ΔT duration. Similar to E, O is also a n dimensional vector with each component O_i of the vector O representing Feature-i received in a particular time interval of testing phase. O is then compared with E using chi-square statistical test. If obtained χ^2 value is less than the predefined threshold, detector accepts H_N. However, if χ^2 value is greater the predefined threshold, detector accepts H_A. Figure 7 shows the steps involved in the detector's operation. For each time interval, detector accepts either H_N or H_A.

VI. Experimental Evaluation

In this section, we describe the experiments conducted to evaluate the detection performance of proposed scheme. In subsequent subsections we discuss testbed architecture, traffic generation for training and testing purpose, detection performance and sensitivity analysis of proposed detection scheme to different significance levels and time interval size.

A. Architecture of Testbed

In order to evaluate the detection performance of proposed technique, we created a testbed similar to the one shown in Figure 8. We designated one computer as web server and connected it to internet. This computer was running Ubuntu 16.04LTS operating system and configured with Apache 2.4.23 software to handle HTTP/2 requests. We chose Apache for our testbed as it is the most popular web server software worldwide [35]. The designated web server was having Intel Core 2 Duo Processor with 4GB of physical memory. We hosted a sample website on this server. This sample website was having 25 web pages containing very brief tutorials of an online course. One out of these web pages was also having an image uploading field to upload an image of size 617 Kilobytes on the web server. This web page was also describing few conditions (such as size and format of image) that should be met for successful uploading of the image. We designated one computer as traffic generator that simulates the behaviour of web users and sends legitimate HTTP/2 traffic towards the web server. This computer was running Linux Mint operating system with AMD Athlon X2 270 Dual core processor and 4 GB of physical memory. We simulated behaviour of web users using a python program. This program triggered execution of 10 threads in parallel, thereby, representing 10 users. Each thread created system calls to start the h2load utility. This utility provides a Console User Interface (CUI) based browser with HTTP/2 support. Using this CUI browser, each thread sent 4 GET requests and 2 POST requests to the server. Using GET requests, a simulated user demanded for different web pages while using POST requests, a simulated user uploaded the image on web server. In order to mimic the real users' web surfing behaviour, we first asked few volunteering users to access the sample website. In particular, they were asked to read the tutorials and upload the image on web server. From this experiment, we then observed the average time delays between each GET and POST request sent by a real user. We found that most users spent more time on a web page having larger size as compared to a smaller web page[3]. This is obvious because of the presence of more information on a larger web page as compared to a smaller one and thus, users need more time to grasp the page contents. We also observed that most of the users spent longest time on web page having image upload field. This was probably due to browsing the valid image file on the local computer and then uploading it to web server. Thus, depending on these observations, we chose appropriate sleeping time intervals ranging from 15 to 25 seconds between each GET request sent by a simulated user. We also chose a longer sleeping time interval of 30 seconds between each POST request sent by a simulated user. Moreover, the simulated users were kept in an uninterrupted loop so that they continued to send HTTP/2 requests unless they were manually terminated. We also designated one computer as malicious client which generated anomalous flows in different time intervals of testing phase. This computer was having 4 GB of physical memory and dual core processor and running Ubuntu 16.04 LTS operating system. Both malicious client and traffic generator were also connected to internet.

B. Traffic Generation for Training

We collected 14 hours of normal HTTP/2 traffic from the web server configured in our testbed and used this traffic for training purpose. We took the time interval size, $\Delta T = 10$ minutes which resulted in 84 intervals in total. We created HTTP/2 normal profile using these 84 intervals. Figure 9 shows the generated normal HTTP/2 profile. The x axis in the figure denotes different features and y axis shows the occurrence count of each of these features. We can notice that except feature-2 (END_STREAM_FLAG reset) remaining features have a count of zero. The feature count for Feature-2 is non zero due to the presence of few HTTP POST requests

[3]This observation is found to be valid in [38] and [37] also.

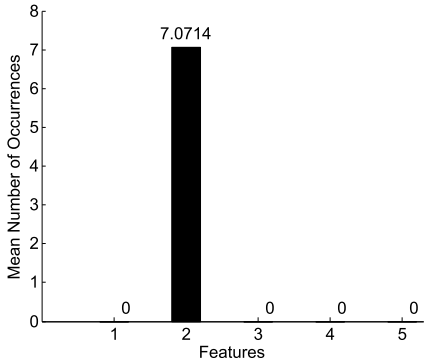

Fig. 9: Normal HTTP/2 Profile

which could not upload the image successfully on the web server and these connections were closed ungracefully by TCP. Such errors in TCP connections over internet are not rare.

C. Traffic Generation for Testing

We generated another 14 hours of normal traffic from the same setup for testing purpose. We repeated this experiment by injecting different attacks at different rates along with normal requests raised by the script. In total there are 7 seven different scenarios (one normal and remaining six containing attack instances) each with 84 intervals. The first scenario has all 84 normal intervals and other six scenarios have 24 intervals without any attack flows and remaining 60 intervals containing different attack flows. Table IX shows experimental scenarios along with number of intervals with various attack flows injected in each case. For example fourth row in the Table indicates that all 7 attacks are injected at the rate of 2 flows per interval. Out of the 60 attack intervals 26 are detected as attack and all 24 normal intervals are detected as normal.

D. Detection Performance

We used time intervals of size $\Delta T = 10$ minutes to assess the detection performance. As mentioned previously, we compute χ^2 statistic value for each time interval using Equation 1 to check for any significant departure of testing profile from the normal one. We can recall that, chi square computation requires dividing the the difference of expected and observed values with expected value. However from the training profile of Figure 9 we can notice that, 4 out of 5 features have a zero count. This results into an expected count of zero for these features and a divide by zero scenario will arise. To avoid this case we increment these feature count by a small number (5 in this case).

We use Recall and False Positive Rate (FPR) as performance metrics to evaluate detection performance of proposed scheme. Recall and FPR are given by Equation 4 and 5 respectively. In these equations, tp = number of attack intervals detected as attack, tn = number of normal intervals detected as normal, fp = number of normal intervals detected as attack, fn = number of attack intervals detected as normal.

$$Recall = \frac{tp}{tp + fn} \tag{4}$$

$$FPR = \frac{fp}{tn + fp} \tag{5}$$

Table IX shows the detection performance of proposed scheme. We obtained these results at 0.05 significance level. If obtained χ^2 values for a particular time interval was greater than the threshold χ^2 value, H_A was accepted for that time interval. However, H_N was accepted in case obtained χ^2 value for that time inerval was less than the threshold χ^2 value.

Our proposed scheme could detect Slow Rate HTTP/2 DoS attacks with high accuracy in case malicious client launched attacks which consume connection pool space for definite amount of time only. However, if we consider anomalous flows of Attack-2 and Attack-4 which have the ability to consume pool for indefinite time also, proposed detection scheme will eventually fail. The reason is malicious client has the option of sending such anomalous flows with very low rate such that traffic profile of the time interval in which attack is launched is almost similar to normal HTTP/2 profile. Thus, obtained χ^2 value is less than the threshold value due to which the time interval is considered as normal, though it contains anomalous flows. In order to detect these cases we can extend the monitoring period for longer duration and count the occurrences of these cases to signal the anomalous scenario.

E. Sensitivity Analysis

The detection performance of the proposed scheme depends on threshold chosen for χ^2 value and also the window period chosen for monitoring various features. Choosing appropriate χ^2 threshold value and size of time interval (ΔT) is a critical and essential task to minimize the false detection cases and increase the true detection cases. In this subsection, we describe the effects of varying threshold χ^2 value and time interval (ΔT) size on the detection accuracy of proposed scheme.

We performed an experiment by varying both threshold χ^2 value and the time interval size ΔT in order to analyze the sensitivity of the proposed scheme to these two parameters. To do so, we generated another 14 hours of traffic by injecting all 7 attacks at the rate of 1 flow/attack after every 300 seconds. We varied the time intervals in step size of 5 minutes ranging from 5 minutes to 25 minutes and took χ^2 values at 7 different significance levels as $\alpha = 0.005, 0.01, 0.025, 0.05, 0.1, 0.25,$ and 0.5. Figure 10 shows that Recall rate increases with increase in both ΔT size and significance level α. We can notice that 100% Recall rate was obtained for $\Delta T = 20$ and 25 minutes independent of α value while worst Recall rate was obtained for $\Delta T = 5$ minutes and $\alpha = 0.005$. Figure 11 shows that FPR remains 0 for $\Delta T = 5$ minutes independent of α value. However, it increases if both ΔT size and α are increased. In our experiments, we obtained worst FPR for $\Delta T = 25$ minutes and $\alpha = 0.5$. This analysis helps us in deciding the appropriate time interval size and threshold significance level. A larger time interval size provides better Recall rate but it also results in higher FPR rate (for larger significance levels) which is undesirable while a smaller time interval size badly affects the Recall rate (for

TABLE IX: Detection Performance of Proposed Scheme

Scenarios	Rate (Flows/attack/interval)	tp	fp	tn	fn	Recall	FPR
Normal	0	0	0	84	0	100.00%	0.00%
Attack-1 and 2	3	5	0	24	55	008.33%	0.00%
Attack-1 to 4	3	8	0	24	52	013.33%	0.00%
All 7 Attacks	2	26	0	24	34	043.33%	0.00%
All 7 Attacks	3	60	0	24	0	100.00%	0.00%
All 7 Attacks	4	60	0	24	0	100.00%	0.00%
All 7 Attacks	5	60	0	24	0	100.00%	0.00%

smaller significance levels) though it results into very small FPR. Thus, choosing appropriate ΔT size and χ^2 value is essential for good detection accuracy of proposed scheme.

F. Periodic Retraining

It should be noted that traffic profile during a busy time interval (belonging to office hour), E_{busy}, may significantly vary from the traffic profile during an idle time interval (night intervals), E_{idle}. As a result, comparing E_{busy} generated during training phase with E_{idle} generated in an interval during testing phase to detect any deviation may lead to higher false detection rates. To handle this non-stationary data distribution scenario, new HTTP/2 traffic profiles can be constructed for different time intervals of a day. For example, two traffic profiles can be generated for a day during training phase - E_{busy} for day time intervals while E_{idle} for night intervals which can then be used for detecting deviations in the traffic received in different time intervals of a day during testing phase. Similarly, different traffic profiles can be generated for week and weekend days. The new traffic profiles can be generated using an online feedback mechanism in proposed scheme. This mechanism enables the detection framework for its periodic retraining. In [31], authors presented a comprehensive study of various schemes which use different online feedback mechanisms to detect anomalies in wireless sensor networks in non-stationary environment. Feedback mechanisms used in these schemes can easily be adapted for periodic retraining of our proposed detection scheme also.

G. Detection with Encrypted HTTP/2 Traffic

The proposed detection scheme requires clear text HTTP/2 payload for further analysis. Thus if HTTP/2 traffic is encrypted, it must be first decrypted before submitting traces to the detector. Nevertheless, this can be easily achieved by implementing an intercepting proxy [43], [9], [28] before the web server designated to handle all HTTP/2 requests. Nowadays, most of the organizations are following this strategy to intercept the traffic coming to and going from their local network. The intercepting proxy server automatically creates SSL certificate used to encrypt/decrpyt traffic between client and proxy server while the original certificate sent by the web server is used to encrypt/decrypt the traffic between proxy and web server. One such example is shown in Figure 12. Using such setup, clear text HTTP/2 traces can be obtained which can then be submitted to detector for further analysis.

VII. Prior Work

In this section, we first discuss prior work related to vulnerabilities present in HTTP/2 protocol and then describe few other schemes proposed in the literature for detecting anomalies even in encrypted network traffic. We also briefly mention some prior works which use chi-square statistical test for anomaly detection.

A. Vulnerabilities in HTTP/2 Protocol

Though several works have been published on DoS attacks against HTTP/1.1 protocol, there are only two works [1], [2] available in the literature which discuss DoS attacks against HTTP/2. Both the works are from the same authors where [2] is an extended version of [1]. In [2], authors demonstrated how legitimate HTTP/2 flash crowd can be launched to create DoS scenario. In [1], behaviour of a web server is presented when it was subjected to large volume of HTTP/2 traffic. In these works, authors used flood of PING and WINDOW UPDATE frames to determine the effect of DDoS attack on the server. The presented approach required a large amount of bandwidth to create DoS scenario (2 million PING frame packets and 2 million $WINDOW\_UPATE$ frame packets), thus, can be easily detected. On the contrary, as mentioned earlier, our proposed Slow Rate HTTP/2 DoS attacks require minimal bandwidth to create DoS scenario and thus, it is much more stealthier than the techniques discussed in [1] and [2]. Table X shows the comparison between the attacks proposed by us and the DDoS attacks presented in [1] and [2] in terms of minimum and maximum number of packets required to cause DoS. To create DoS scenario using attacks proposed in this paper, the minimum and maximum number of packets required are 150 (against Apache) and 2060 (against Nginx) respectively. On the other hand, DoS attacks discussed in [1] and [2] require millions of packets. Also, authors did not mention which web servers were tested against the DDoS attacks during their experiments. Another limitation with the attacks proposed in [1] and [2] is that these attacks could only achieve Reduction of Quality (RoQ) scenario instead of DoS as they were not able to exhaust the computational resource at web server completely.

Imperva [24], a security consultancy firm, found four flaws in HTTP/2 protocol- *Slow Read*, *HPACK (Compression)*, *Dependency DoS* and *Stream abuse*. They tested these attacks against five different web servers and showed that all web servers were vulnerable to at least one attack vector. To create DoS scenario by exploiting first flaw, authors first set very small SETTINGS_INITIAL_WINDOW_SIZE and sent

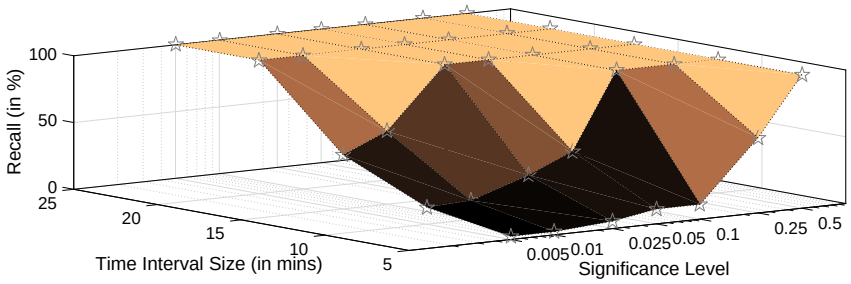

Fig. 10: Recall

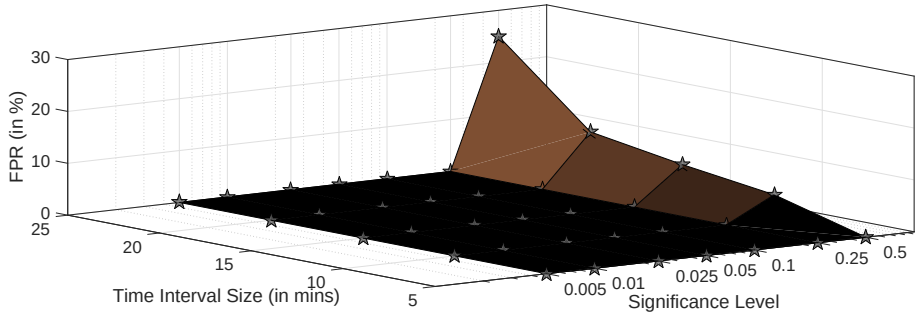

Fig. 11: False Positive Rate

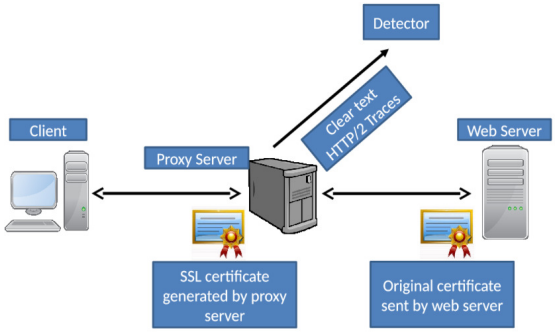

Fig. 12: Intercepting HTTP/2 traffic using Proxy Server

TABLE X: Comparison of Proposed Attacks with the Attacks discussed in [1] and [2]

	Minimum Number of Packets Required	Maximum Number of Packets Required
Proposed Attacks	150	2,060
Attacks in [1]	2,000,000	2,000,000
Attacks in [2]	1,000,000	2,000,000

GET requests from several concurrent streams multiplexed over a single TCP connection. Servers like Apache 2.4.17 and Apache 2.4.18 are vulnerable to the attack because these software versions dedicate one thread per stream which results into consumption of all worker threads once enough number of streams are created. However, Apache 2.4.20 and later server versions are not vulnerable as these versions dedicate threads on per connection basis instead of per stream basis. The second flaw was related to HPACK compression algorithm used by HTTP/2 protocol to compress header size. For compression purpose, sender informs other endpoint about the maximum size of header compression table which can be used to decode header blocks. The encoder, then can select any table size less than or equal to this value. However, RFC 7540 does not put a restriction on the size of each individual header. Thus, size of each header is restricted to the size of header compresion table. To create DoS using second flaw, authors generated a first stream with a large header equal to the size of table. After this, multiple streams were created over the same connection and each of them referenced to the single large header multiple number of times. As a result, server kept allocating memory to decompress each request and once the whole memory was consumed, legitimate clients could not access the server. Third flaw involved exploitation of stream dependency and priority implementations where authors created scenarios like dependency cycle and/or rapid changes in the stream dependencies. Authors observed that this lead to high CPU utilization and unreasonable memory utilization. The last flaw involved reusing stream identifiers over the same connection which RFC 7540 strictly prohibits. However, server implementations like Microsoft's Internet Information Services (IIS) earlier did not follow this mechanism and as a result, the web server was found vulnerable to this attack. Nevertheless, all these flaws are now patched with release of latest updates from each vendor [24].

Yahoo penetration testing team, in their work [22], presented

some flaws found in various HTTP/2 implementations like Firefox, Apache Traffic Server (ATS) and Node-http2. These flaws basically caused issues like arbitrary code execution, unreasonable memory allocation due to integer underflow and buffer out of bound reads.

B. Detecting Anomalies in Encrypted Network Traffic

Various schemes which detect anomalies in network traffic by observing hidden periodicities and hard-to-masquerade properties such as inter-packet arrival time, time gap between a request and its corresponding response, etc. can detect the proposed attack (and any other Slow Rate DoS attacks) as number of requests with no corresponding responses is very high in presence of proposed attacks. One such technique is proposed in [4] where authors collect information such as duration of a request and its corresponding response, the time gap between the end of a request and the start of the relative response and the time gap between the end of a response and the next request on the same established connection. Similar detection scheme is proposed in [29] which analyzes specific spectral features of traffic over small time horizons. In particular, proposed scheme tracks the number of packets received by a web server in a particular time period. In [3] also, authors proposed a scheme that monitors the number of packets received by a web server in different time horizons for anomaly detection. Since these anomaly detection techniques do not require payload analysis, they are effective in case of encrypted traffic also. Nevertheless, to determine which of the five Slow Rate DoS attacks proposed in this paper are launched and using what method, analyzing payload is the only possible solution.

C. Chi-Square Test

Chi-square test has been vastly used to detect network anomalies. Authors in [39], [41], [42], [40] proposed detection mechanisms which use chi-square test to distinguish normal events from intrusive events. In this paper, events like password guessing and attempts to gain an unauthorized remote access are considered as intrusive events. Authors also showed that the χ^2 test detected intrusive events with good accuracy. Similar detection mechanism that uses χ^2 test was proposed in [18] also. This detection mechanism checks if the network is under attack and then, based on the initial observation, deploys accurate filtering rules. These filtering rules minimize the attack impact.

VIII. Conclusion

In this paper, we proposed novel Slow Rate DoS attacks against HTTP/2 services. These attacks can be launched by sending specially crafted HTTP/2 payloads to a web server. We performed an empirical evaluation of these attacks against all major web servers and found that majority of them are vulnerable to these attacks independent of clear text or encrypted HTTP/2 requests used to launch the attacks. After a comparative study of HTTP/1.1 and HTTP/2 for Slow Rate DoS attacks we conclude that HTTP/2 has more threat vectors than HTTP/1.1. Subsequently, we proposed an anomaly detection technique to detect the proposed attacks which works by measuring the χ^2 value between the expected and observed traffic pattern and showed that it could detect the attacks with high accuracy. We believe that this work will trigger further assessments of HTTP/2 protocol which may lead to finding of new vulnerabilities and mitigating of these threat vectors with appropriate fixes.

References

[1] E. Adi, Z. Baig, C. P. Lam, and P. Hingston. Low-Rate Denial-of-Service Attacks against HTTP/2 Services. In *2015 5th International Conference on IT Convergence and Security (ICITCS)*, pages 1–5, 2015.

[2] E. Adi, Z. A. Baig, P. Hingston, and C. Lam. Distributed Denial-of-Service Attacks Against HTTP/2 Services. *Cluster Computing*, 19(1):79–86, 2016.

[3] M. Aiello, E. Cambiaso, M. Mongelli, and G. Papaleo. An On-line Intrusion Detection Approach to Identify Low-rate DoS Attacks. In *2014 International Carnahan Conference on Security Technology (ICCST)*, pages 1–6, 2014.

[4] M. Aiello, E. Cambiaso, S. Scaglione, and G. Papaleo. A Similarity based Approach for Application DoS Attacks Detection. In *2013 IEEE Symposium on Computers and Communications (ISCC)*, pages 430–435. IEEE, 2013.

[5] M. Aiello, G. Papaleo, and E. Cambiaso. SlowReq: A Weapon for Cyberwarfare Operations. Characteristics, Limits, Performance, Remediations. In *International Joint Conference SOCO13-CISIS13-ICEUTE13*, pages 537–546. Springer, Cham, 2014.

[6] S. Batthalla, M. Swarnkar, N. Hubballi, and M. Natu. VoIP Profiler: Profiling Voice over IP User Communication Behavior. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 312–320, 2016.

[7] M. Belshe, R. Peon, and M. Thomson. (RFC 7540) Hypertext Transfer Protocol Version 2 (HTTP/2), 2015.

[8] Application-Layer DDoS Attacks Are Growing: Three to Watch Out For. http://www.breakingpointsystems.com/resources/blog/application-layer-ddos-attacks-growing/.

[9] Burp Proxy. https://portswigger.net/burp/.

[10] E. Cambiaso, G. Papaleo, and M. Aiello. Slowcomm: Design, Development and Performance Evaluation of a New Slow DoS Attack. *Journal of Information Security and Applications*, 35:23–31, 2017.

[11] E. Cambiaso, G. Papaleo, G. Chiola, and M. Aiello. Slow DoS Attacks: Definition and Categorisation. *International Journal of Trust Management in Computing and Communications*, 1(3-4):300–319, 2013.

[12] E. Cambiaso, G. Papaleo, G. Chiola, and M. Aiello. Designing and Modeling the Slow Next DoS Attack. In *International Joint Conference: CISIS'15 and ICEUTE'15*, pages 249–259. Springer, Cham, 2015.

[13] Is It Statistically Significant? The Chi-square Test. http://www.ox.ac.uk/media/global/wwwoxacuk/localsites/uasconference/presentations/P8_Is_it_statistically_significant.pdf.

[14] CVE-2016-1546. Denial-of-Service by Thread Starvation.

[15] T. Dierks and E. Rescorla. (RFC 5246) The Transport Layer Security (TLS) Protocol Version 1.2, 2008.

[16] Y. Dodge. *The Concise Encyclopedia of Statistics*. Springer Science & Business Media, 2008.

[17] C. Douligeris and A. Mitrokotsa. DDoS Attacks and Defense Mechanisms: Classification and State-of-the-art. *Computer Networks*, 44(5):643–666, 2004.

[18] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred. Statistical Approaches to DDoS Attack Detection and Response. In *Proceedings of DARPA Information Survivability Conference & Exposition, 2003. DISCEX'03.*, volume 1, pages 303–314. IEEE, 2003.

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. (RFC 2616) Hypertext Transfer Protocol – HTTP/1.1, 1999.

[20] D. Golait and N. Hubballi. Detecting Anomalous Behavior in VoIP Systems: A Discrete Event System Modeling. *IEEE Transactions on Information Forensics and Security*, 12(3):730–745, 2017.

[21] h2load: Benchmarking Tool for HTTP/2 and SPDY Server. https://nghttp2.org/documentation/h2load.1.html.

[22] Attacking HTTP/2 Implementations. https://yahoo-security.tumblr.com.

[23] N. Hubballi and N. Tripathi. A Closer Look into DHCP Starvation Attack in Wireless Networks. *Computers & Security*, 65:387 – 404, 2017.

[24] HTTP/2: In-depth Analysis of the Top Four Flaws of the Next Generation Web Protocol. https://www.imperva.com/docs/Imperva_HII_HTTP2.pdf.

[25] jNetPcap. http://jnetpcap.com/docs/javadocs/jnetpcap-1.4/index.html.

[26] G. Macia-Fernandez, J. E. Diaz-Verdejo, and P. Garcia-Teodoro. Mathematical Model for Low-Rate DoS Attacks Against Application Servers. *IEEE Transactions on Information Forensics and Security*, 4(3):519–529, 2009.

[27] G. Mantas, N. Stakhanova, H. Gonzalez, H. H. Jazi, and A. A. Ghorbani. Application-layer Denial of Service Attacks: Taxonomy and Survey. *International Journal of Information and Computer Security*, 7(2-4):216–239, 2015.

[28] mitmproxy. https://mitmproxy.org/.

[29] M. Mongelli, M. Aiello, E. Cambiaso, and G. Papaleo. Detection of DoS Attacks through Fourier Transform and Mutual Information. In *2015 IEEE International Conference on Communications (ICC)*, pages 7204–7209. IEEE, 2015.

[30] OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/source/.

[31] Colin O'Reilly, Alexander Gluhak, Muhammad Ali Imran, and Sutharshan Rajasegarar. Anomaly Detection in Wireless Sensor Networks in a Non-stationary Environment. *IEEE Communications Surveys & Tutorials*, 16(3):1413–1432, 2014.

[32] J. Park, K. Iwai, H. Tanaka, and T. Kurokawa. Analysis of Slow Read DoS Attack. In *2014 International Symposium on Information Theory and its Applications (ISITA)*, pages 60–64. IEEE, 2014.

[33] T. Peng, C. Leckie, and K. Ramamohanarao. Survey of Network-based Defense Mechanisms Countering the DoS and DDoS Problems. *ACM Computing Surveys*, 39(1), 2007.

[34] scapy-ssl_tls 1.2.2. https://pypi.python.org/pypi/scapy-ssl_tls/1.2.2.

[35] Netcraft: September 2016 Web Server Survey. https://news.netcraft.com/archives/2016/09/19/september-2016-web-server-survey.html.

[36] N. Tripathi and N. Hubballi. Exploiting DHCP Server-side IP Address Conflict Detection: A DHCP Starvation Attack. In *2015 IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS)*, pages 1–3, 2015.

[37] N. Tripathi, N. Hubballi, and Y. Singh. How Secure are Web Servers? An Empirical Study of Slow HTTP DoS Attacks and Detection. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 454–463, 2016.

[38] T. Yatagai, T. Isohara, and I. Sasase. Detection of HTTP-GET flood Attack Based on Analysis of Page Access Behavior. In *2007 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 232–235, 2007.

[39] N. Ye and Q. Chen. An Anomaly Detection Technique based on a Chi-square Statistic for detecting Intrusions into Information Systems. *Quality and Reliability Engineering International*, 17:105–112, 3 2001.

[40] N. Ye, S. M. Emran, Q. Chen, and S. Vilbert. Multivariate Statistical Analysis of Audit Trails for Host-based Intrusion Detection. *IEEE Transactions on Computers*, 51(7):810–820, 2002.

[41] N. Ye, S. M. Emran, X. Li, and Q. Chen. Statistical Process Control for Computer Intrusion Detection. In *Proceedings of DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01.*, volume 1, pages 3–14. IEEE, 2001.

[42] N. Ye, X. Li, Q. Chen, S. M. Emran, and M. Xu. Probabilistic Techniques for Intrusion Detection based on Computer Audit Data. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 31(4):266–274, 2001.

[43] OWASP Zed Attack Proxy Project. https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

[44] S. T. Zargar, J. Joshi, and D. Tipper. A Survey of Defense Mechanisms against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys & Tutorials*, 15(4):2046–2069, 2013.