Effective Analysis of Secure Web Response Time

Carlos López[1], Daniel Morato[1,2], Eduardo Magaña[1,2], Mikel Izal[1,2]
[1] Public University of Navarre, Department of Electrical, Electronic Engineering and Communications, Pamplona, Spain
[2] Institute of Smart Cities, Pamplona, Spain
email: {carlos.lopezromera, daniel.morato, eduardo.magana, mikel.izal}@unavarra.es

***Abstract*—The measurement of response time in web based applications is a common task for the evaluation of service responsiveness and the detection of network or server problems. Traffic analysis is the most common strategy for obtaining response time measurements. However, when the traffic is encrypted, the analysis tools cannot provide these measurement results. In this paper we propose a methodology for measuring the response time in HTTPS traffic based on the flow of data in each direction. We have validated the tool with real traffic and with a worst case scenario created in a testbed. When pipelining is present in the encrypted HTTP 1.1 traffic, it results in a small error in the measurement (between 5% and 15% of error for the 99.9 percentile of the real response time). However, pipelining support has almost disappeared from modern web browsers; this makes the estimation provided by this methodology very accurate in real traffic measurements, even for low probability response times. More than 98.8% of the over 8.6 million request-response times we measured in our campus Internet link were obtained without any error.**

***Index Terms*—HTTP, HTTPS, traffic analysis, APM, response time**

I. Introduction

HyperText Transfer Protocol (HTTP) is a widely used internet protocol, employed not only on the public World Wide Web. It is the underlying mechanism for fundamental corporate management applications like ERPs (Enterprise Resource Planning) or CRMs (Customer Relationship Management). The new offerings of these services are web-based, and even cloud-based. High responsiveness from these applications is critical for an effective company workforce.

Application Performance Monitoring (APM) is an important task in network and service management. It offers tools for the detection of performance problems, such as server memory limitations, high CPU utilization or network congestion. They can be measured directly on the servers, on the network switches or links or they can be indirectly obtained by monitoring parameters such as application response times, which offer a quantitative measure of the client's perceived quality of service. Isolated high response times do not signify an actual problem; however, repetitive high latency values might be indicative of a pathology concerning a server or an application.

Performance problems can be detected from server logs [1]. Nonetheless, this information is incomplete, and it can be insufficient when dealing with an array of issues [2], [3]. A more complete view of application behaviour can be obtained by passively monitoring network traffic. This allows the discovery of issues associated to the underlying transport protocols, such as those related to reliable transport, congestion control or flow control. Passive traffic monitoring is common in business environments using port mirroring techniques [4], [5].

Nowadays, with increasing HTTPS adoption [6], HTTP requests and responses are concealed, which poses a problem to application performance monitoring - request-response times are no longer measurable. Some APM tools offer traffic decryption [7], [8], [9], [10]. However, decryption (which already introduces an additional layer of processing when analyzing traffic) requires access to the server's private key, which may not be possible in all scenarios. For example, monitoring office workers' traffic towards external web applications (such as cloud based ERPs) is not possible using this technique. Moreover, with the arrival of version 1.3 of Transport Layer Security (TLS), perfect forward secrecy becomes mandatory [11], which means that decryption is not possible even in possession of the server's private key, due to the ephemeral keys generated for each connection.

In this paper, we evaluate a technique for HTTPS response time analysis, without traffic decryption, which allows anomaly detection. We present the results obtained from employing this method on a large dataset, as well as the possible issues that arise when analyzing HTTP. The results show high accuracy in obtaining an estimation of the frequency of high response times for corporate access link traffic.

The rest of the paper is organized as follows. Section II describes the measurement algorithm and the characteristics from HTTP and TLS it depends on. Section III presents the scenarios where network traffic was collected in order to evaluate the accuracy in the measurement. Section IV contains the results comparing the estimation to the ground truth measurement and section V concludes the paper.

II. HTTPS analysis

HTTP is an application level protocol which has received considerable attention from the research community since the late 90s [12]. Its secure counterpart, HTTPS, has been commonly deployed since privacy in personal data or economic transactions became frequent in the Web [13].

Response time analysis for HTTP or HTTPS requests can be easily accomplished from server access logs [1], [3], [14]. However, server logs hide client perceived response times. Client side results can be obtained based on active response time measurements [15], however they are not extensible to

This work was supported by Spanish MINECO through project PIT (TEC2015-69417-C2-2-R).

978-3-903176-17-1 / © 2019 IFIP

a large population of users. When server logs are not easily accessible or they do not provide all the necessary information, service analysis is accomplished based on passively-monitored network traffic. The research literature on HTTP traffic analysis is abundant [2], [14], [16], [17], [18]; however, HTTPS limits its effectiveness due to payload encryption.

HTTPS response time analysis from network traffic can be based on TLS session data decryption [9], [10], [19]. However, in most networking scenarios, decryption is not possible, and a blind traffic analysis is the only option. There are only a few previous works on blind HTTPS performance analysis. Most papers restrict their scope to macroscopic measurements or to some kind of user, browser, operating system or web site fingerprinting [20], [21], [22]. In [23] the authors present a tool which is able to measure response times for HTTPS traffic based on the analysis of the distribution of packet sizes and arrival times. However it requires a variable threshold parameter to distinguish container object requests (such as the main HTML document of a web page) and its embedded objects. It also needs to record every client's round-trip time (RTT) and it is not perfectly robust to parallel download, resulting in accuracy loss. With pipelining present, they work on the assumption that non-pipelined requests will be within a size range, which might not be true for every environment in which HTTP is used. Overall, size-based analysis does not seem to be flexible enough to work in settings where the nature of requests and responses might be different than those of the test context.

A. HTTPS fundamental mechanics

The TLS [11] protocol has an initial handshake phase, where certificates are possibly exchanged, a common encryption suite is negotiated, and encryption keys are generated. This stage is performed in clear text, and it can be analyzed by any packet inspection tool. The encryption keys cannot be obtained from the traffic but these steps in session establishment can be easily monitored. From there on, both parties send encrypted Application Data messages to one another. These HTTPS messages contain the original HTTP header and payload. APM tools try to measure the response times to these HTTP requests. The requested URL cannot be decrypted but an anonymous response time could be extracted from network traffic.

A client performs a single initial HTTPS handshake in a TCP transport connection, which establishes the encryption session (see Fig. 1). HTTP 1.1, transported over TLS, offers persistent connections, i.e. a single connection can be used for multiple HTTP requests. The TLS handshake is not part of the HTTP request, and many requests can take place after a single session establishment. These negotiated parameters can even be shared with other TCP connections established in the following minutes between the same peers. This initial handshake time affects the response time experienced by the user. However, it can be measured independently, and be taken into account if it indicates an issue. In this paper, we focus on measuring request-response times for encrypted data, since analyzing the plaintext handshake phase is a trivial task.

B. Analysis methodology

HTTP analysis can be accomplished based on per-packet, mostly stateless, techniques [24] or it can require the reconstruction of both TCP streams in the connection. The analysis techniques without reconstruction are faster but they are also error prone in the presence of packet losses. TCP stream reassembly is a very common task in APM traffic analysis tools and stateful firewalls [25]. It is necessary before TLS decryption can be applied. We relax the requirement of stream decryption in the analysis process but we do keep the continuous application data streams as an input to the algorithm. Therefore, losses, disordered packets, and retransmissions do not pose a handicap for the analysis. Previous papers have shown that TCP stream reconstruction is feasible for traffic rates of several gigabits per second using multi-core processing architectures [25], [26].

Reconstructed stream data can be available for analysis as soon as the TCP sequence is continuous. We are not assuming stream reconstruction after a connection finishes but a live reconstruction. The analysis module sees new data available as an application using a TCP socket would see it. As soon as the TCP stack in the host (or in the reconstruction module) has new in-sequence bytes available they are offered to the application (or to the analysis tool). We have implemented and tested a TCP stream reconstruction module for passive traffic analysis.

From the reconstructed TCP streams we can extract TLS Application Data messages. These messages contain the HTTP Protocol Data Units (PDUs). Several Application Data messages can be required in order to send a large HTTP request or response. A simple HTTP GET request is usually contained in a single Application Data message, however, an HTTP POST message request, uploading a large file, could require several messages. The responses to these messages also require one or more TLS Application Data messages depending on content length. These messages will be contained in one or more TCP segments and therefore IP packets.

Our proposed method looks at Application Data message bursts in each direction[1]. It assumes a burst is equivalent to an HTTP request or response, depending on its direction (client to server or vice versa). A request ends when the first byte from an Application Data message from server to client is available. The response ends when the connection is closed or when the first byte of an Application Data message from client to server is available. The beginning of a response marks the end of the request and the beginning of a request marks the end of the response to the previous one.

Fig. 1 shows an example of an HTTPS session. After the TCP connection establishment and TLS session management, a bidirectional full-duplex encrypted stream is available. A request can be contained in several Application Data messages,

[1] The source code for this algorithm is available on request

which in turn can be contained in several TCP segments. TCP segmentation is hidden to the APM tool by the TCP stream reconstruction function. All the Application Data messages from client to server are assigned to a single request, which ends when new data from server to client is measured (the response). The end of the response is marked by new data from client to server.

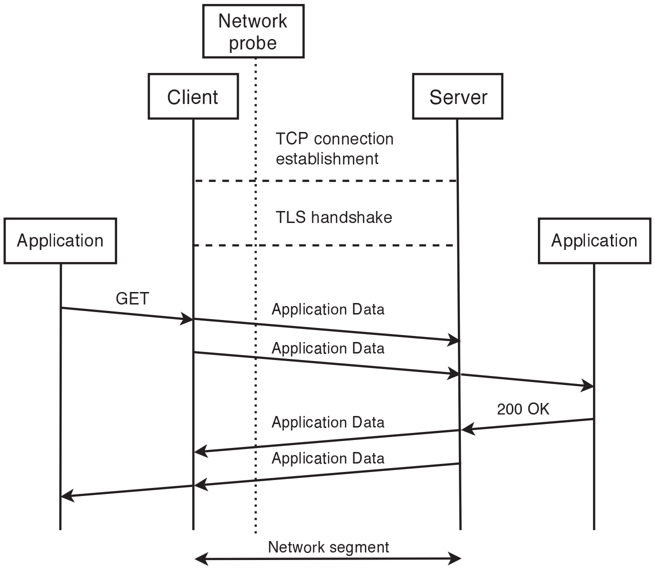

Fig. 1. Timeline of events in an HTTPS session

C. Pipelining and HTTPS analysis

The proposed analysis methodology requires that only one standing request exists per connection. This is the behaviour present in HTTP 1.0, however, HTTP 1.1 explicitly allows pipelining. This means that several HTTP requests can be sent from client to server, previous to the response to any of them. The protocol only requires that responses keep the same order as the requests. This ordering results in Head-of-Line (HoL) blocking in the HTTP 1.1 stream.

Pipelining breaks the request-response sequence, therefore it can cause problems in the analysis. The HTTP pipelining mechanism could lead the analysis tool to consider several requests as a single one. A burst of several requests in a row cannot be distinguished from a single request due to encryption. Also, the responses to pipelined requests will be considered a single response. This means that employing the proposed method in the presence of pipelining could yield higher response times overall, and that it will report a lower number of requests, due to the grouping. Fig. 2 shows an example where the second HTTP request is sent before the response to the first request arrives. The APM tool will consider both TLS messages as part of a single request and the responses as part of a single response.

If the pipelined requests are sent back-to-back (without waiting for any response), the response time given by the analysis tool should be a good approximation of the highest response time in the pipelined burst. The highest response time is expected to take place for the last request in the pipelined burst, due to HoL blocking (its response time will be increased by all the previous responses in the pipeline).

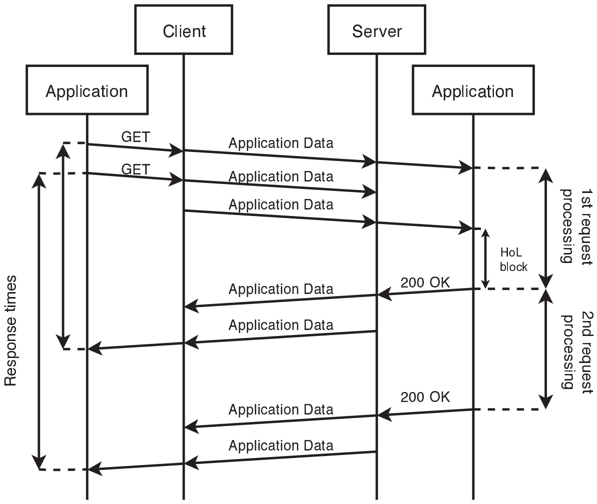

Fig. 2. Pipelining in HTTP 1.1

In the following sections we show the results obtained from measuring response times using the proposed methodology and we compare them to the real response times that could be measured by decrypting the streams.

III. Measurement and analysis scenarios

We ran the proposed APM tool for several traffic traces. We used a large traffic trace from our university campus Internet link, and several traffic traces obtained from a testbed scenario. We created the testbed in order to control the amount of pipelining present in the traffic. These traces provide a scenario where the effect of pipelining on accuracy can be measured.

For evaluation purposes, HTTP traffic was used instead of its encrypted counterpart. Although the main purpose of this paper is proposing a method to analyze encrypted web traffic, verification of its accuracy using HTTPS traffic requires the decryption of the TLS streams, which is not always possible. We adapted the algorithm to analyse HTTP traffic. Instead of waiting for Application Data messages, it takes any TCP data from client to server as part of a request and any traffic from server to client as part of a response. The end of a request is marked by data from the server (the beginning of the response) and the end of the response is marked by the end of the connection or by new application level traffic from client to server (the beginning of a new request). Being HTTP traffic, we can use an HTTP traffic analysis tool in order to provide the ground truth response time measurement. We developed this tool, based on the same TCP stream reconstruction module used by the HTTPS APM tool.

A. Testbed scenario

As previously explained, traffic was captured from a testbed scenario to control and see the effect of pipelining on the proposed traffic analysis method. To do so, software scripts were developed to recreate clients connecting to a server with a controllable pipelining parameter. This parameter establishes the largest pipelining burst possible in a connection. For each new burst, a random number is drawn from a discrete uniform distribution between 2 and the largest desired number of requests in the burst. These requests are sent back-to-back from client to server and the client waits for the responses before initiating a new random burst.

For each request, following previous measurements on web traffic [12], a random response time was determined from a pareto distribution with $\alpha = 1.21$ and decay after $x \approx 0.3$ seconds[2].

The testbed machine uses Apache 2.4.34 with the prefork Multi-Processing Module (MPM) [27] as the HTTP server. The client scripts create several concurrent TCP connections to the server, each one representing a user or one of the parallel connections between browser and server.

We want to control the amount of pipelining in order to measure its effect on the accuracy of response time measurements. We do not need to model parameters such as client reading time, connection duration or response size, since they have no effect on response times to individual requests.

Each experiment generated 2,000,000 HTTP requests. Table I shows the number of bursts in each experiment, which is approximately the total number of requests divided by the average burst length. Using HTTP analysis, all the 2 million response times were measured while the blind traffic analysis provides only one measurement per pipelining burst.

TABLE I
PARAMETERS IN PIPELINING BURST GENERATION (NUMBER OF HTTP REQUESTS PER BURST) AND NUMBER OF BURSTS IN THE TESTBED EXPERIMENTS

Maximum burst size	**Average burst size**	**Number of pipelining bursts**
2	2	1044808
3	2.5	832866
5	3	518387
10	6	314851
15	8.5	239172
20	11	177532
20	11	177532
25	13.5	150270
30	16	127078

B. Real world scenario

We verified the analysis method using also a large web traffic trace from our university campus Internet access link. The trace was collected during more than 4 days (from January 24th 2019 15:26 to January 29th 2019 13:26). It contains 5.3 million TCP connections with HTTP traffic from more than 78,000 users. These users include local campus users accessing public web servers and remote users accessing campus servers. The number of different users is obtained by the number of different client IP addresses. The average number of requests per connection was 1.6, with a total 8.6 million HTTP request-response pairs. HTTPS traffic represents a larger percentage of link usage. We can run the APM tool using the HTTPS traffic, however we cannot validate the response times it provides, therefore we proceeded to the validation using HTTP traffic.

IV. Tool validation results

A. Testbed scenario

We measure response times from the beginning of a request to the end of the response. Using HTTP dissection we decode HTTP headers and obtain the ground truth measurement. Using the developed APM tool we obtain estimated response time values. We cannot compare the values on a one-on-one basis, as the APM tool provides fewer results than the real number of HTTP requests. However, we are interested in detecting repetitive performance issues, which even in a sampled measurement can be detected from the probability of extreme response time values.

Fig. 3 shows the survival function for the cumulative probability distribution of response time, obtained using HTTP dissection or using the APM tool. It presents the results for several degrees of pipelining, described by the average pipelining burst length. In a burst, each subsequent request to the first one presents a response time which includes the response time from the previous requests (see Fig. 2). Therefore, the higher the average burst length the heavier the distribution tail is. The results offered by the APM tool follow the distribution shape from the ground truth, however, it offers a worst case estimation (larger probability values $P(T_{resp} > t)$). The reason is that it takes only one measurement from each burst, which corresponds approximately to the response time for the last request in the burst.

The results offered by the APM tool are closer to the real response time values when the average burst length is small. We checked the quality in the estimation for percentiles 99 and 99.5. Fig. 4 shows the percentage of error in the estimation. For an average burst length larger than 3 requests the estimation is within at least 80% of the desired value (less than 20% error). For lower burst lengths it improves to 85%, but more important, it does not get worse as the average pipelining burst size increases, even when using runs of 30 requests or more. For the 99.9 percentile the quality of the estimation is even better, and less than 10% error is obtained for bursts shorter than 8 requests.

We must highlight that this error in the estimation is due to the presence of pipelining. It would not exist in case of no pipelining. We are evaluating how the quality of the estimation depends on the degree of pipelining *assumming pipelining is present.* We show on the following section that pipelining is not so common nowadays.

As previously discussed, in presence of pipelining, we expect the response time value measured by the HTTPS APM tool to be very similar to the worst response time in each pipelined burst - typically the last request in it, due to HoL blocking. To test this, the largest response time for each burst was extracted. Fig. 5 shows the survival function for

[2] These traffic traces are available on request

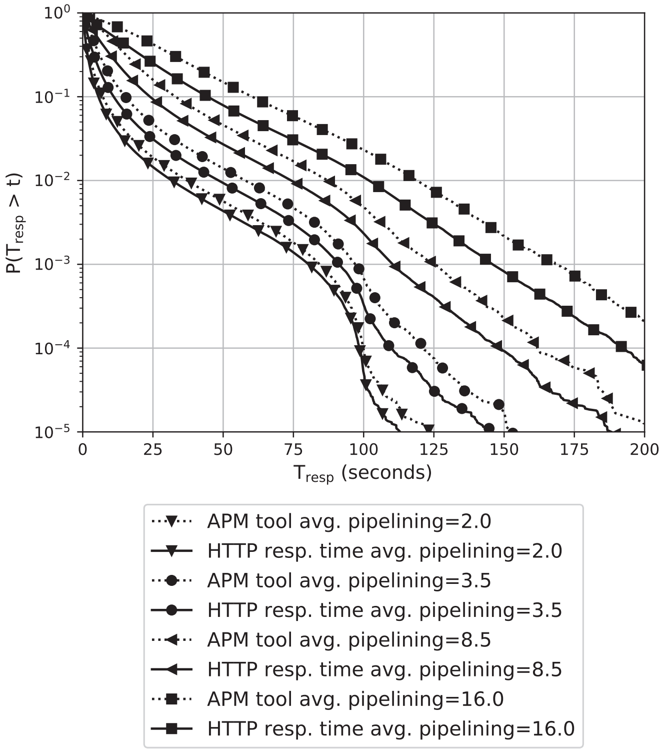

Fig. 3. Survival function for the cumulative probability function of response time and APM results (only a few distributions are shown for better clarity)

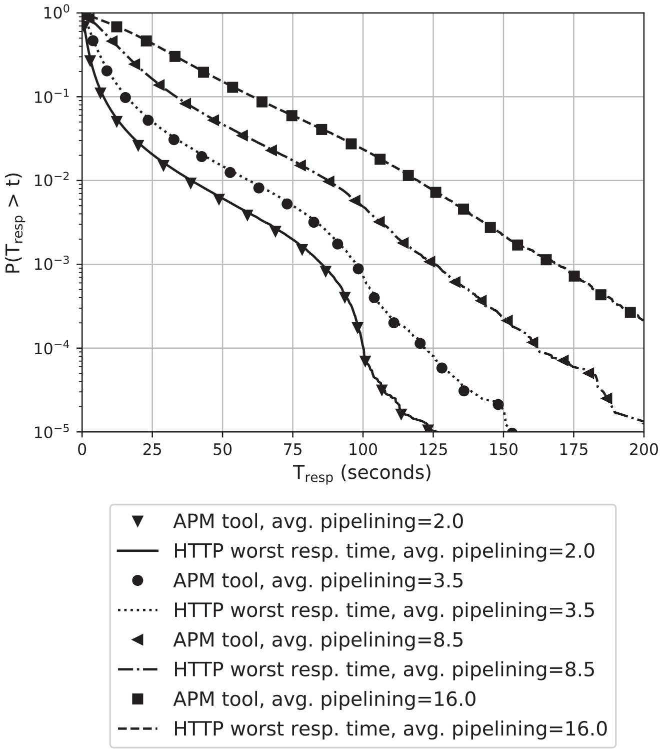

Fig. 5. APM tool results compared to the largest response time per burst

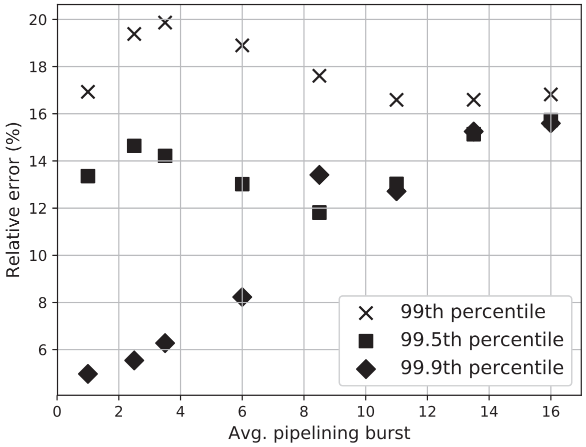

Fig. 4. Percentage of error in the estimation of percentiles in the cumulative distribution function of response time

the cumulative probability distribution of the largest response time per burst (HTTP dissection), compared to the results from the APM tool. Both are nearly identical, which validates the hypothesis.

B. Real world traffic analysis results

The testbed has revealed that the quality of the response time estimation depends heavily on the presence and degree of pipelining in the traffic. The real HTTP traffic trace we described in section III-B contains 5.3 million connections, however, less than 3,000 of them present pipelining, i.e. less than 0.06% of the connections. More than 8 million request-response pairs take place in connections without pipelining, therefore for more than 99.11% of the requests there should be no error in the estimation. Those 3,000 connections present an average burst length of 6.73 requests and a maximum of 10 requests in the pipeline.

Fig. 6 shows the survival function for the cumulative probability distribution of response time, obtained using HTTP dissection or using the APM tool with the real traffic trace. Visually, the results match with high precision for probabilities above 10^{-4}.

In fact, automated analysis of each individual connection shows that 98.855% of the over 8.6 million request-response times match perfectly between the results from both the HTTP and HTTPS APM tools.

Worse results were expected, but they were conditioned to the presence of pipelining. However, for some time now, the mechanism of pipelining has been abandoned in web browsers due to bugs, poor retrocompatibility with older servers, inconsistent behaviour with proxies and HoL blocking [28]. While pipelining presented a solution to high latency environments, where sending several requests would save round trips, the main browsers have discarded its use (or disabled it by default) in favor of solving this issue with the coming of HTTP/2.0 [29], [30]. The main User-Agents we found sending pipelined requests were APT (a software package manager for GNU/Linux), Android browsers with User-Agent Dalvik, and

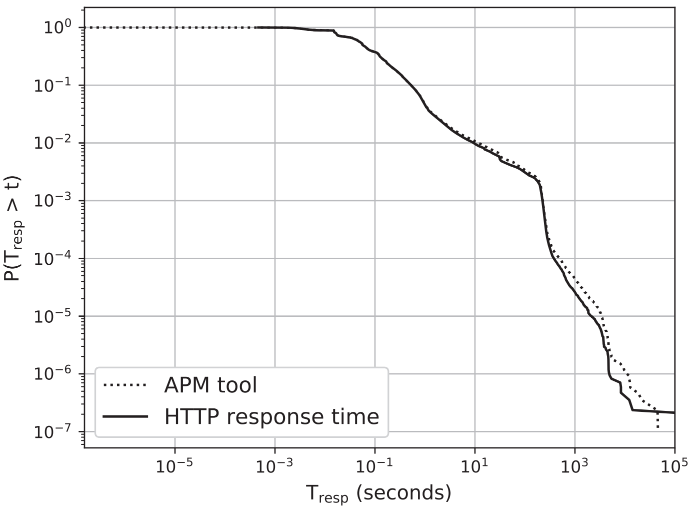

Fig. 6. APM tool and ground truth comparison for the real traffic trace

Apple iPads.

However, we acknowledge that pipelining might be present in specific environments dominated by proprietary software that uses different HTTP libraries. Some common libraries that support pipelining are Net::Async::HTTP [31] for Perl, Twisted [32] for Python, Apache HttpCore [33], Microsoft .NET Framework [34], and QNetworkRequest [35] and libcurl [36] for C++.

C. Protocol features which affect the analysis

During the implementation and debugging of an HTTP dissector for obtaining ground truth measurements we detected several other protocol features which could deviate the results obtained using the APM tool:

- Error response codes: Some HTTP error response codes are sometimes sent from the server before the whole request is received. This is the case for errors 400 (Bad Request), 413 (Request Entity Too Large), 414 (Request-URI Too Long) or 500 (Internal Server Error). A large request, requiring several TCP segments, can result in the server sending an error response before the whole request has been sent. The APM tool cannot see the error code as it is encrypted (blind analysis), therefore the response breaks the request packet flow and the last segments of the request are taken as a new request.
- 100 Continue: When a client has a large request body to send, it has the option to send first an HTTP header with the *Expect: Continue* option. The server may respond with an *HTTP 100 Continue* code to signify that the following request body will be accepted, based on the received header. This allows the client to check whether its request will be accepted before sending the actual request, thus saving bandwidth if such request would be denied. As the response code is encrypted, it cannot be recognised and the request-response sequence is broken into two.
- Keep-Alives: Some TCP implementations send TCP Keep-Alive segments to prevent the connection from being closed after an idle period. They send one garbage byte at the end of the TCP stream sequence. Although this byte should be considered a retransmission, it elects an HTTP 1.1 400 Bad Request response in some servers.
- WebSockets: Using the WebSocket protocol an HTTP session becomes a two-way communication channel where each end-point can send data independently. WebSockets break the request-response behaviour, therefore the measured response times using the described APM tool would be wrong. In our 5.3 Million HTTP connections only 766 of them offered the option to upgrade to the WebSocket protocol. This is less than 0.02% of the connections, hence we do not expect a large deviation in the response time distribution due to errors in measurement for this protocol.

Although the analysis was accomplished using HTTP traffic, all the features described above are expected to be present in HTTPS traffic. There are no HTTPS-specific mechanisms that could result in an erroneous measurement. All the above described situations are due to HTTP protocol features.

D. Other advantages over HTTP dissection

We implemented the dissector while expecting HTTP traffic to be RFC-compliant. However, we found several situations where the dissector required heuristics in order to decode some server responses. Some of these situations come from HTTP servers being non-conformant to HTTP 1.1 RFC [37], while some others have their origin in bad programming techniques at the server side. We detail the most significant situations we found:

- HEAD attached to a body: HTTP HEAD requests result in a response which must not [37] contain a body, even if the response specifies a *Content-length*. The length specified is for the body that would be sent in case for example of a GET request. We have found some servers that send the body of the response even for a HEAD request. An RFC-conformant client will not expect any body in the response, therefore it will not read from the TCP stream after the HTTP header was complete (based on a blank line). The body of the response stays in the input TCP stream and it will be read by the client when it expects the answer to another request, causing confusion to the HTTP decoder.
- Excessive body: We have found some situations where the HTTP response contains the *Content-length* field with a number of bytes specified for the body which do not agree with the real number of bytes sent after the HTTP header. Usually, the body is larger than the size specified in the header field. The reason is a server script which forces the value in the *Content-length* field but afterwards it sends more bytes than it announced. It can be for example a PHP script that sends a file to the client but some error in the script creates a text error message, which is sent through the output stream to the HTTP response, adding

more bytes to the response, which were not accounted for in the header. In other situations the server-side script sends a footer, maybe because the developer didn't notice that the footer was included in all the scripts and this footer adds more bytes to the body than the *Content-length* size that was announced.
- Malformed headers: HTTP 1.1 RFC specifies that header lines end with a pair or characters CRLF (Carriage-Return Line-Feed), however, there are server implementations which use only the line-feed character. The RFC also provides the names for the header options, but we found servers that use illegal option names, meaning that instead of a *Connection* option they wrote "Coennction" or maybe "nnCoection", which makes recognising a header option a difficult task. The source of these error are probably simple servers, used in embedded systems. These servers can become quite common with the increase in IP-based Internet of Things (IoT) devices.

In these situations, an RFC-compliant analysis will result in errors due to illegal HTTP responses. However, the APM tool we have described does not carry any deep header analysis, therefore it cannot fail in these situations and it provides the correct measurement without any additional heuristic.

A blind analysis can provide results where a deep analysis would fail without adding some heuristics.

E. Applicability to HTTP/2 traffic

The algorithm presented in this paper can also be applied to HTTP/2 traffic over TCP or over Quick UDP Internet Connections (QUIC).

In HTTP/2, due to the protocol's inherent stream multiplexing [38], packet bursts can no longer be assumed to contain a single request or response. TCP packets can contain frames from several HTTP messages, which in an encrypted stream are indistinguishable. In fact, using HTTP/2, servers can even push resources to the clients before a request has been issued. We expect a decreased accuracy of blind analysis due to these features.

As for HTTP/2 over QUIC, which has been proposed as HTTP/3 [39], the differences introduced by QUIC in the traffic's characteristics remain to be tested, yet they are expected to have a lesser impact compared to those from HTTP/2.

These two scenarios have been left for future work.

V. Conclusions

In this paper we have proposed the analysis of response time for encrypted HTTPS requests based on the data flow between endpoints. The procedure overestimates the correct value when HTTP pipelining is present, however, support of pipelining has been eliminated from most major browsers, making its presence insignificant in the traffic we collected from our university campus access link (more than 5 million connections). When considering the whole set of requests in a network link, the proposed methodology provides a good approximation of the distribution of response times, even for probabilities as low as 10^{-4}. It also simplifies the analysis when the server does not follow the requirements in HTTP protocol syntax.

References

[1] Thiam Kian Chiew and Karen Renaud. Estimating web page response time based on server access log. In *2015 9th Malaysian Software Engineering Conference (MySEC)*, 2014.

[2] Anja Feldmann. BLT: Bi-layer tracing of HTTP and TCP/IP. *Computer Networks: The International Journal of Computer and Telecommunications Networking archive*, 33(1-6):2476–2490, June 2000.

[3] J. Rexford B. Krishnamurthy. Software issues in characterizing web server logs. In *World Wide Web Consortium Workshop on Web Characterization*, Nov 1998.

[4] BIG-IP System: Implementing a passive monitoring configuration. https://support.f5.com/content/kb/en-us/products/big-ip_ltm/manuals/product/bigip-passive-monitoring-configuration-13-0-0/_jcr_content/pdfAttach/download/file.res/BIG-IP_System__Implementing_a_Passive_Monitoring_Configuration.pdf, accessed on April 8th, 2019.

[5] Network Visibility - Ixia. https://www.ixiacom.com/solutions/network-visibility, accessed April 8th, 2019.

[6] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. Measuring https adoption on the web. In *26th USENIX Security Symposium*, 2017.

[7] TLS/SSL decryption & encryption - Ixia. https://www.ixiacom.com/products/tls-and-ssl-decryption-and-encryption, accessed April 8th, 2019.

[8] SSL monitoring - DC RUM 12.4. https://community.dynatrace.com/community/display/DCRUM124/SSL+monitoring, accessed April 8th, 2019.

[9] Bernd Greifeneder, Bernhard Reichl, Helmut Spiegl, and Gunter Schwarzbauer. Method of non-intrusive analysis of secure and non-secure web application traffic in real-time, 2003. US Patent 7,543,051 B2.

[10] Doron Kolton, Adi Stav, Asaf Wexler, Ariel Ernesto Frydman, and Yoram Zahavi. System to enable detecting attacks within encrypted traffic, 2006. US Patent 7,895,652 B2.

[11] E. Rescorla. The Transport Layer Security (TLS) protocol version 1.3, 2018. RFC 8446.

[12] Azer Bestavros Mark E. Crovella, Murad S. Taqqu. *A practical guide to heavy tails*, chapter 8, pages 3–25. Birkhauser Boston Inc., 1998.

[13] Prasant Mohapatra Udaykiran Vallamsetty, Krishna Kant. Characterization of E-commerce traffic. *Electronic Commerce Research*, 3(1-2):167–192, January-April 2003.

[14] Mehul Nalin Vora and Dhaval Shah. Estimating effective web server response time. In *2017 Second International Conference on Information Systems Engineering (ICISE)*. IEEE, April 2017.

[15] Ludmila Cherkasova, Yun Fu, Wenting Tang, and Amin Vahdat. Measuring and characterizing end-to-end internet service performance. *ACM Transactions on Internet Technology*, 3(4):347–391, November 2003.

[16] Paul Barford and Mark Crovella. Critical path analysis of TCP transactions. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication - SIGCOMM '00*. ACM Press, 2000.

[17] Chang-Gyu Jin and Mi-Jung Choi. Integrated analysis method on HTTP traffic. In *2012 14th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, September 2012.

[18] Xiuwen Sun, Kaiyu Hou, Hao Li, and Chengchen Hu. Towards a fast packet inspection over compressed HTTP traffic. In *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, June 2017.

[19] Dmytro Ageyev Maxim Tawalbeh Vitalii Bulakh Tamara Radivilova, Lyudmyla Kirichenko. Decrypting SSL/TLS traffic for hidden threats detection. In *Proceedings of IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, 2018.

[20] Martin Husák, Milan Čermák, Tomáš Jirsík, and Pavel Čeleda. HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting. *EURASIP Journal on Information Security*, 6:1–14, Feb 2016.

[21] Jonathan Muehlstein, Yehonatan Zion, Maor Bahumi, Itay Kirshenboim, Ran Dubin, Amit Dvir, and Ofir Pele. Analyzing HTTPS encrypted traffic to identify users operating system, browser and application. In *Proceedings of the 14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, 2017.

[22] Alfonso Iacovazzi, Andrea Baiocchi, and Ludovico Bettini. What are you Googling? - inferring search type information through a statistical classifier. In *Proceedings of IEEE Global Communications Conference (GLOBECOM)*, 2013.

[23] Jianbin Wei and Cheng-Zhong Xu. Measuring client-perceived pageview response time of internet services. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):773–785, 2011.

[24] Carlos Vega, Paula Roquero, and Javier Aracil. Multi-Gbps HTTP traffic analysis in commodity hardware based on local knowledge of tcp streams. *Computer Networks*, 113(11):258–268, Feb 2017.

[25] Kai Zhang, Junchang Wang, Bei Hua, and Xinan Tang. Building high-performance application protocol parsers on multi-core architectures. In *Proceedings of IEEE 17th International Conference on Parallel and Distributed Systems*, Dec 2011.

[26] Jing Xu, Hanbo Wang, Wei Liu, and Xiaojun Hei. Towards high-speed real-time HTTP traffic analysis on the Tilera many-core platform. In *Proceedings of IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, June 2013.

[27] Apache MPM Prefork. https://httpd.apache.org/docs/2.4/mod/prefork.html, accessed on May 15th, 2019.

[28] Mark Nottingham. Making HTTP pipelining usable on the open web. Internet-Draft draft-nottingham-http-pipeline-01, IETF Secretariat, March 2011. http://www.ietf.org/internet-drafts/draft-nottingham-http-pipeline-01.txt.

[29] Firefox 54: Changes for web developers - MDN web docs. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/54#HTTP, accessed April 8th, 2019.

[30] The Chromium Projects: HTTP pipelining. https://www.chromium.org/developers/design-documents/network-stack/http-pipelining, accessed April 8th, 2019.

[31] Net::Async::HTTP documentation - metacpan.org. https://metacpan.org/pod/Net::Async::HTTP#pipeline-=\%3E-BOOL, accessed April 8th, 2019.

[32] Twisted: HTTPChannel class documentation. https://twistedmatrix.com/documents/8.2.0/api/twisted.web2.channel.http.HTTPChannel.html#lingeringClose, accessed April 8th, 2019.

[33] Apache HttpCore Examples - Apache HttpComponents. https://hc.apache.org/httpcomponents-core-ga/examples.html, accessed April 8th, 2019.

[34] HttpWebRequest.Pipelined Property (System.Net) - Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/api/system.net.httpwebrequest.pipelined?view=netframework-4.7.2, accessed April 8th, 2019.

[35] QNetworkRequest class - Qt documentation. https://doc.qt.io/Qt-5/qnetworkrequest.html, accessed April 8th, 2019.

[36] CURLMOPT_PIPELINING explained - cURL documentation. https://curl.haxx.se/libcurl/c/CURLMOPT_PIPELINING.html, accessed April 8th, 2019.

[37] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, June 2014. RFC 7231.

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

[39] Mike Bishop. Hypertext Transfer Protocol Version 3 (HTTP/3). Internet-Draft draft-ietf-quic-http-17, IETF Secretariat, December 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-http-17.txt.