Design and Evaluation of HTTP Protocol Parsers for IPFIX Measurement

Petr Velan, Tomáš Jirsík and Pavel Čeleda

Institute of Computer Science, Masaryk University, Brno
Czech Republic, {velan|jirsik|celeda}@ics.muni.cz

Abstract. In this paper we analyze HTTP protocol parsers that provide a web traffic visibility to IP flow. Despite extensive work, flow meters generally fall short of performance goals due to extracting application layer data. Constructing effective protocol parser for in-depth analysis is a challenging and error-prone affair. We designed and evaluated several HTTP protocol parsers representing current state-of-the-art approaches used in today's flow meters. We show the packet rates achieved by respective parsers, including the throughput decrease (performance implications of application parser) which is of the utmost importance for high-speed deployments. We believe that these results provide researchers and network operators with important insight into application visibility and IP flow.

Keywords: HTTP, protocol, parser, traffic, measurement, flow, IPFIX

1 Introduction

Flow monitoring technologies, such as NetFlow or IPFIX, are widely used in large-scale networks to provide situational awareness. They provide information about *who* communicates with *whom*, *when*, *how long*, using *what protocol* and *service* and also *how much data* was transferred. Acquired flow data is based on IP headers (network and transport layer) and it does not include any payload information. On the other hand, we observe that HTTP protocol [6] became a *"new Transmission Control Protocol"* (TCP). More and more applications rely on HTTP protocol, e.g. Web 2.0 content, audio and video streaming, instant messaging etc. HTTP traffic (TCP port 80) can usually pass through most firewalls and therefore presents a standard way of transporting/tunneling data. The versatility, ubiquity and amount of HTTP traffic makes it easy for an attacker to hide malicious activities. Missing application layer visibility renders standard NetFlow and IPFIX to be ineffective for HTTP monitoring.

Network and security devices use application layer analysis to provide application visibility, monitoring and traffic control. For example, Cisco Application Visibility and Control (AVC) [4] solution uses next-generation deep packet inspection (NBAR2) and flexible NetFlow to identify, classify and report on over 1,000 applications. HTTP information elements are supported by YAF [8] and

nProbe [5] flow meters and are exported in IPFIX format. Most intrusion detection systems extract application layer data for in-depth analysis.

Deep Packet Inspection (DPI) predominates for application layer analysis. While it is possible for small and medium sized networks to effectively deploy the DPI, the amount of traffic in large (10+ Gb/s) networks makes the inspection of every packet a challenging problem. The performance and results that can be achieved depend on a number of factors including flow meter configuration and analyzed traffic distribution. Due to the complexity and sheer number of application protocols, it is hard to compare different environments/platforms and derive conclusions on which solution is the best.

Our research attempts to answer following question: *What are the impacts of application layer analysis of HTTP protocol on flow meters and flow monitoring process?* The contribution of our work is threefold: *(i)* We designed and evaluated several HTTP protocol parsers representing current state-of-the-art approaches used in today's flow meters. *(ii)* We introduced a new flex-based HTTP parser. *(iii)* We report on the throughput decrease (performance implications of application parser) which is of the utmost importance for high-speed deployments.

The paper is organized as follows. Section 2 describes related work. Section 3 contains a description of the HTTP inspection algorithms and the framework that was used to test the algorithms. Section 4 describes the methodology used for HTTP parsers performance comparison. Section 5 presents the performance evaluation of the individual algorithms. Finally, Section 6 contains our conclusions.

2   Related Work

Application layer protocol parsers are an integral part of many network monitoring tools. We explored the source code of the following frameworks to see how the HTTP parsing is implemented. nProbe uses standard glibc [2] functions like *strncmp* (compare two strings) and *strstr* (locate a substring). YAF uses Perl Compatible Regular Expressions (PCRE) [1] to examine HTTP traffic. Suricata [14] and Snort [18] are both written in C. Suricata uses LibHTP [17] library which does HTTP parsing using custom string functions while Snort does its parsing using glibc functions. httpry [3] is another HTTP logging and information retrieval tool which is also written in C and uses its own built-in string functions. These HTTP parsers are hand-written.

Other approach is taken by Bro [16] authors. They use binpac [15], a declarative language and compiler designed to simplify the task of constructing robust and efficient semantic analyzers for complex network protocols. They replaced some of Bro existing analyzers (handcrafted in C++) and demonstrated that the generated parsers are as efficient as carefully hand-written ones.

In this paper, we try to determine whether these approaches to HTTP parsing can handle large traffic volumes. Besides the above approaches, we propose to use the Fast Lexical Analyzer (Flex) [11] to design a new HTTP parser. Flex

converts expressions into a lexical analyzer that is essentially a deterministic finite automaton that recognizes any of the patterns. The algorithm that converts a regular expression directly to deterministic finite automaton is described in [10] and [13].

There are other works that inspect the HTTP protocol headers. In [19] the authors use statistical flow analysis to differentiate traditional HTTP traffic and Web 2.0 applications. In [21] the authors identify HTTP sessions based on flow information. In both cases a ground truth sample is needed, which is a topic addressed by [22]. In [7] and [12] the authors use DPI to obtain information from the HTTP headers. Our approach is orthogonal to these works, since we are interested in extending IP flow records with HTTP data.

3   Parser Design

HTTP protocol [6] has a number of properties that can be monitored and exported together with IP flow data. The most commonly monitored ones are present in almost every HTTP request or response header. Based on the properties monitored by the state-of-the-art DPI tools we selected the following ones for our parsers: *HTTP method, status code, host, request URI, content type, user agent and referer*. Keeping track of every bidirectional HTTP connection is too resource consuming on high speed networks, thus we focus on evaluation of each individual packet. This approach is more common for flow meters since it is more resistant to resource depletion attacks.

We implemented and evaluated three different types of parsing algorithms. The first algorithm (*strcmp* approach) loops the HTTP header line by line and searches each line for given fields. It uses standard glibc string functions like *memchr*, *memmem* and *strncmp*. The simplified pseudocode is shown in Algorithm 1. The second algorithm (*pcre* approach) uses several regular expressions taken from YAF to search the packet for specific patterns indicating HTTP header fields. The pseudocode for the *pcre* algorithm is shown in Algorithm 2. We designed the third algorithm (*flex* approach) to handle each packet as a long string. It uses finite automaton to find required HTTP fields and the Flex lexer is used to process the packets. The automaton design is shown in Fig. 1.

Since the Flex is a generic tool, its initialization before scanning each packet is quite an expensive operation. Therefore we decided to remove all unnecessary dynamic memory allocations and costly initializations to see whether the performance can be increased. We named the new version *optimized flex*. The disadvantage of Flex is that it has to keep the data in its own writable buffer. Therefore the received data must be copied to such a buffer, which adds to the processing costs significantly. The advantage of the *flex* parser is its simple maintenance and extension possibilities. The framework can be modified to parse any other application layer protocol just by changing the set of regular expression rules. The *strcmp* parser would have to be rewritten from a scratch.

The *strcmp* implementation also offers a space for further improvement. Algorithm 3 shows an *optimized strcmp* version of the code that features a better

Algorithm 1 *strcmp*

1: **if** first line contains "HTTP" **then**
2:    **while not** end of HTTP header **do**
3:       **for** every parsed HTTP field **do**
4:          **if** field matches the line **then**
5:             store the value of the line
6:          **end if**
7:       **end for**
8:       move to the next line
9:    **end while**
10:    **return** HTTP packet
11: **else**
12:    **return not** HTTP packet
13: **end if**

Algorithm 2 *pcre*

1: **if** first line contains "HTTP/x.y" **then**
2:    **for all** PCRE rules **do**
3:       **if** rule matches **then**
4:          store the matched value
5:       **end if**
6:    **end for**
7:    **return** HTTP packet
8: **else**
9:    **return not** HTTP packet
10: **end if**

Fig. 1: *flex* algorithm schema

processing logic. The optimized version searches for specific strings by comparing several bytes at once, which is done by casting the character pointer to integer pointer. The number that is compared to the string is computed from ASCII codes of the characters and converted to network byte order. The size of the used integer depends on the length of the string; longer integers offer better performance.

To focus only on the HTTP parsing algorithms we decided to let the FlowMon exporter [9] handle the packet preprocessing. We used a benchmarking (input) plugin that reads packets from PCAP file to memory at start-up. Then it supplies the same data continuously for further processing. This approach allows us to focus on benchmarking the algorithms without the necessity of considering the disk I/O operations. We provide the source code of implemented algorithms and used packet traces at the paper homepage [20].

Algorithm 3 *optimized strcmp*

1: **if** payload begins with "HTTP" **then**
2:     store status code
3:     **while not** end of HTTP header **do**
4:         **for** every parsed response HTTP field **do**
5:             **if** line starts with field name **then**
6:                 store the value of the line
7:             **end if**
8:         **end for**
9:         move to the next line
10:     **end while**
11:     **return** HTTP response packet
12: **end if**
13: **if** payload begins with one of GET, HEAD, POST, PUT,
            DELETE, TRACE, CONNECT **then**
14:     store request URI
15:     **while not** end of HTTP header **do**
16:         **for** every parsed request HTTP field **do**
17:             **if** line starts with field name **then**
18:                 store the value of the line
19:             **end if**
20:         **end for**
21:         move to the next line
22:     **end while**
23:     **return** HTTP request packet
24: **end if**
25: **return** not HTTP packet

---

4   Evaluation Methodology

In this section we define a methodology of HTTP protocol parsers evaluation. We focus on parsing performance (number of processed packets per second) of the algorithms described in Section 3 from several different perspectives.

The first perspective focuses on the performance comparison with respect to analyzed traffic structure. The second perspective covers the impact of the number of HTTP fields supported by parser. The third perspective describes the effect of a Carriage Return (CR or '\r') and a Line Feed (LF or '\n') control characters distribution in packet payload.

A common technique of increasing network data processing performance is processing only important part of each packet. Therefore, we perform each of the tests in two configurations. In the first configuration the parsers are given whole packets. This is achieved by setting limit on packet size to 1500 bytes, which is the most common maximum transmission unit value on most Ethernet networks. In the second configuration the parsers are provided with truncated packets of length 384 bytes, which is the minimum packet length recommended for DPI by authors of the YAF exporter [8].

To test the performance of the parsers, we created an HTTP traffic trace (testing data set). Our requirements on the data set were as follows: preserve

the characteristics of HTTP protocol, reflect various HTTP traffic structures and have no side effects on the flow meter. In order to meet these requirements we decided to create synthetic trace.

The HTTP protocol is a request/response protocol. To preserve the characteristics of HTTP protocol during the testing a random request, response and binary payload packet was captured from the network. To omit the undesirable bias of the measurement only these three packets were used to synthesize test trace. The final test trace consists of 200 packets. In order to reflect various traffic structures, we suggested following ratio:

$$r = \frac{\#\text{request packets} + \#\text{response packets}}{\#\text{all packets}} * 100 \tag{1}$$

where $r \in [\,0, 100\,]$ and created a test set for each integer ratio from the interval. Further, we created two packets with modified payload. One packet contained the CR and LF control characters only at the very *beginning* of the packet payload, the other one only at the *end*. For both of the modified packets and for the *unchanged* packet the test trace for each integer ratio was created.

Having defined the test trace, we propose the following case studies to cover all evaluation perspectives. The case studies are carried out for both full and truncated packets. Moreover, we measure the performance of the flow meter without an HTTP parser (*no HTTP* parser). This way we can estimate the performance decline caused by increased application layer visibility.

1. *Performance Comparison*: This case study compares the parsing performance of implemented parsers. Moreover, we report on the flow meter performance without an HTTP parser (*no HTTP* parser).
2. *Parsed HTTP Fields Impact*: This case study shows a parser performance with respect to the number of supported HTTP fields. We incrementally add support for new HTTP fields and observe the impact on the parser performance.
3. *Packet Content Effect*: The result of this study presents the influence of the CR and LF control characters position in a packet payload on the parser performance. The test traces containing modified payload packets are used to perform the measurement.

The performance evaluation process employs the benchmarking input plugin (see Section 3) to obtain the number of processed packets per second. In order to avoid influencing the results, the plugin uses separate thread and CPU core for the accounting. The plugin counts the number of the processed packets in ten second interval and then computes the packets per second rate. We have operated the benchmark plugin for fifty seconds for each test trace and computed a number of packets processed and a standard error of the measurement. The parsed HTTP header fields impact and packet content effect was assessed in a similar way. All measurements were conducted on a server with the following configuration: Intel Xeon E5410 CPU at 2.33 GHz, 12 GB 667 MHz DDR2 RAM and Linux kernel 2.6.32 (64 bit).

5   Parser Evaluation

In this section, we present results of HTTP parser evaluation. First we describe the parser performance comparison, then we investigate the impact of supported HTTP header fields. Finally, the effect of the packet content on HTTP parsing performance is shown.

Performance Comparison. This case study uses the standard version of each parser that supports seven HTTP fields. The data set containing the unmodified payload packets is used and the parsers are tested both on full and truncated packets. Fig. 2 shows the result for full packets case study and Fig. 3 shows performance evaluation for truncated packets.

First we discuss the Fig. 2. The *no HTTP* meter is capable of parsing more than 11 million packets per second. This result is not influenced by the application data carried in the packet, since the data is not accessed by the *no HTTP* parser. Employing event the fastest of the HTTP parsing algorithms the performance drops to the nearly one half of parsed packets per second. All of the HTTP parsers show the decrease in the performance as the ratio r increases since the amount of request and response packets, which are more time demanding to parse, grows.
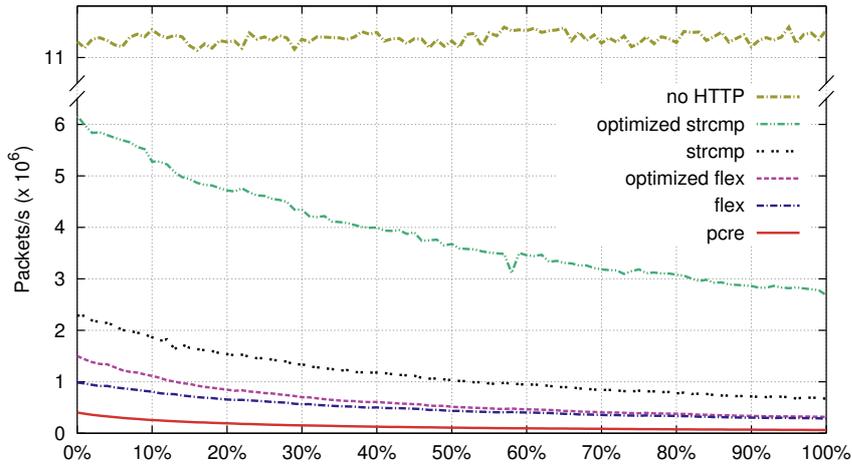

Fig. 2: Parser performance comparison with respect to HTTP proportion (0 % - no HTTP, 100 % - only HTTP headers) in the traffic - full packets 1500 B.

The best performance is achieved by *optimized strcmp* parser, which uses application protocol and code level optimizations. The parser takes into account the HTTP header structure, the difference between HTTP request and response

headers and looks only for header fields that can be found in the specific header type. The code level optimizations include converting static strings into integers and matching them against several characters at once, which can be done in one processor instruction. The *strcmp* parser performance is the second best, although the throughput is less than half of the *optimized strcmp* parser.

The main difference between *flex* and *optimized flex* parsers is in the automaton initialization process. By rewriting the initialization process we achieved slight performance improvement, which is noticeable mainly in the $\langle 0\,\%, 20\,\% \rangle$ interval, where the actual HTTP parsing time is short. There is one other important factor affecting the *flex* parser performance. The flex automaton is designed to work with its own writable buffer, since it marks end of individual parsed tokens directly into the buffer. For this reason a copy of the packet payload must be created before the actual parsing can start. To measure the impact of the copying, we created another two parser plugins called *empty* and *copy*. First we measured the flow meter throughput with *empty* plugin which performs no data parsing, then with *copy* plugin which only copies packet payload to static buffer. From the results we estimate the throughput the *optimized flex* parser would have without the memory copying. The performance of the *optimized flex* parser would be about 2.4 million packets per second for 0 % and 0.33 million packets per second for 100 % HTTP packets. This shows that the actual HTTP parsing, when compared to *strcmp* parser, is slightly faster for binary payload packets and slower for HTTP header packets.

The performance of the *pcre* parser is the lowest. The PCRE algorithm converts the regular expression to a tree structure and then performs a depth-first search while reading the input string. In case there is no match in current tree branch, the algorithm backs up and tries another one. Therefore, for a complex regular expression the pattern matching is not that fast as simple string search using functions like *strcmp*. Another reason why the *pcre* parser is not fast is that it performs all searches on whole packet payload. The other algorithms are processing the data sequentially.

Fig. 3 shows the results for truncated packets. The *optimized strcmp* and *no HTTP* are only slightly faster since the truncating of the packets has positive impact on CPU data cache utilization. The *strcmp* algorithm is flawed since its throughput on HTTP packets deteriorates rapidly. This shows the disadvantage of hand-written parsers, as they are more error-prone than the generated ones. The *pcre* parser performance is almost doubled, as the repeatedly processed data are truncated. Flex based parser also achieve performance increase, since the memory copying costs are reduced for smaller data.

Parsed HTTP Header Fields Impact. This case study was designed to show the impact of a number of parsed HTTP header fields on the parser performance.

When payload packets are detected, they do not have their content parsed for additional HTTP header fields. Therefore, a test set containing only HTTP request and response packets was used. The case study starts with an empty plugin, that does not parse HTTP header fields and just labels the HTTP pack-
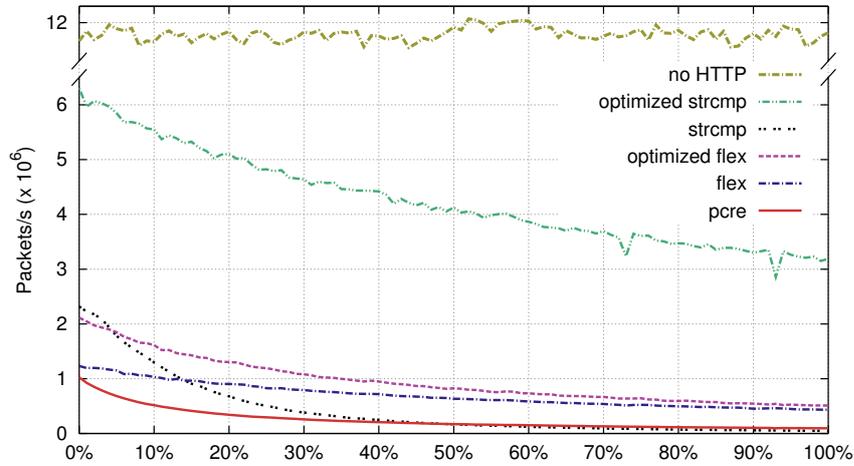
Fig. 3: Parser performance comparison with respect to HTTP proportion (0 % - no HTTP, 100 % - only HTTP headers) in the traffic - truncated packets 384 B.

ets. In the next steps we cumulatively add an additional header field to parse until we parse all of the seven supported fields. We run the tests for both full and truncated packets. The average performance of the parsers for each of the added field is shown in Fig. 4.

Only the request and response packets are parsed, thus the values for the seven fields parsed in the Fig. 4 correspond to the 100 % packet/s values in the Fig. 2 and Fig. 3. For the same reason the parsed packets per second numbers are lower in comparison with the Fig. 2 and Fig. 3. The performance of *strcmp* and *pcre* parsers drops with each additional parsed HTTP header field. The *optimized strcmp* parser implementation details attentional fluctuation affect on performance shown in Fig. 4. An example is the performance increase when adding a *(4) request URI* or a *(3) status code*. It is caused by extra code snippet that extracts the URI so that this line is not processed by the more generic code designed for parsing other header fields. Due to the usage of finite automaton the data is always processed in one pass by the flex-based algorithms. Therefore, they retain the same level of performance for all additional fields. This feature could be used to automatically build powerful parsers, when the large number of parsed application fields would make it ineffective to create hand-written parsers.

Same as in the previous case study, the parsers processing truncated packets show better performance than the parsers working on full packets.

Packet Content Effect. This case study investigates the possible effects of the position of the CR and LF control characters in the packet payload on the parser performance. The mentioned ASCII characters represent the end of line in the HTTP header. Some of the proposed algorithms use these characters as
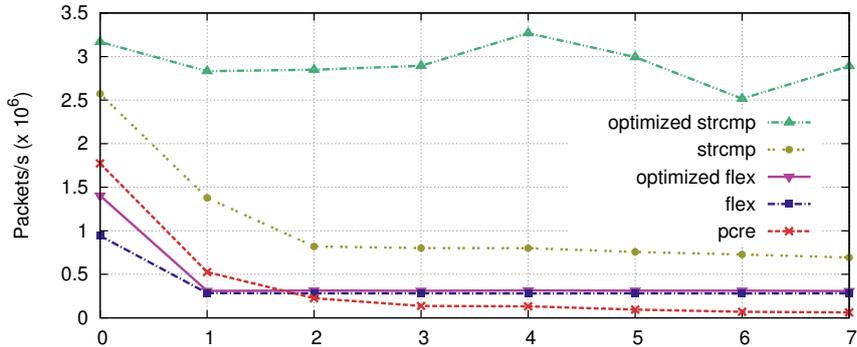
Fig. 4: An HTTP parser throughput for 1500 B packets; supported fields - (0) *none* - HTTP protocol labeling, (1) *+host*, (2) *+method*, (3) *+status code*, (4) *+request URI*, (5) *+content type*, (6) *+referer*, (7) *+user agent*.

the trigger to stop parsing. Therefore the position of these characters affects the parsers performance. The packets with the CRLF characters at the very *beginning* should be parsed faster than the packets having the CRLF at the *end* since the algorithm terminates as soon as it identifies the CRLF characters. The test sets with modified binary payload packets (see Section 4) enables us to compare the algorithms taking in account this perspective.
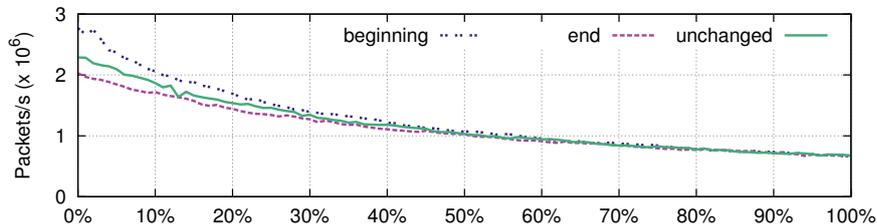

Fig. 5: Packet content effect - packet length 1500 B.

We used the modified binary payload packets to test the parsers. The parsing algorithms, except the *strcmp* algorithm, show insignificant difference in their performance for all variants of the modified packets. The *pcre* and *optimized strcmp* parsers do not search for end of line characters in order to label the packet, therefore this test does not affect them. The flex-based algorithms are not significantly affected since they stop parsing on first character that is not expected in HTTP header and therefore stop at the first character in any case. The *strcmp* parser depends on the search for end of line characters, which is confirmed by Fig. 5. The sooner the characters are found, the faster the algorithm terminates.

The scenario with truncated packets is different, since the performance on *end* data set is greater than on *unchanged* data. This is caused by removing the end of packet payload together with the end of line character. When the *strcmp* algorithm cannot find the character, it terminates immediately without trying to search the data. Therefore it terminates sooner than on *unchanged* data set, where the end of line character is found and the search continues.

6   Conclusions

This paper has assessed the impacts of HTTP protocol analysis on flow monitoring performance. We implemented the state-of-the-art approaches to HTTP protocol parsing. Moreover, the new flex-based HTTP parser was designed and its performance was compared to the other approaches.

The evaluation shows that in our case the hand-written and carefully optimized parser performs significantly better than implementations with automated parsing. It also shows that the new flex-based implementations handles the increasing number of parsed HTTP fields without significant performance loss. Truncating the packets prior to HTTP protocol parsing can increase the parser throughput. The performance comparison of *no HTTP* parser with HTTP parsers shows that providing an application visibility is a demanding task. Current approaches to the application protocol parsing may not be effective enough to process a high-speed network traffic.

Although we focused on HTTP header parsing in this paper, measuring overall performance of flow meters is also essential. We will address performance evaluation and runtime requirements of entire flow meter frameworks in our future work. This research will allow us to compare existing frameworks and new prototypes under equal conditions. Monitoring HTTP application protocol expose new challenges for flow meters. In particular, an increased number of exported fields, large flow record length and their impact on transport protocol requires further research.

References

1. PCRE - Perl Compatible Regular Expressions (Nov 2012), `http://www.pcre.org/`
2. The GNU C Library (glibc) (Dec 2012), `http://www.gnu.org/software/libc/`
3. Bittel, J.: httpry - HTTP logging and information retrieval tool (Apr 2013), `http://github.com/jbittel/httpry`
4. Cisco Systems, Inc.: Application Visibility and Control (Apr 2013), `http://www.cisco.com/go/avc`
5. Deri, L.: nProbe: an Open Source NetFlow probe for Gigabit Networks. In: In Proc. of Terena TNC 2003 (2003)

6. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard) (Jun 1999), `http://www.ietf.org/rfc/rfc2616.txt`, updated by RFCs 2817, 5785, 6266, 6585

7. Gehlen, V., Finamore, A., Mellia, M., Munafò, M.M.: Uncovering the big players of the web. In: Proceedings of the 4th international conference on Traffic Monitoring and Analysis. pp. 15–28. TMA'12, Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-28534-9_2`

8. Inacio, C.M., Trammell, B.: YAF: Yet Another Flowmeter. In: Proceedings of the 24th international conference on Large installation system administration. pp. 1–16. LISA'10, USENIX Association, Berkeley, CA, USA (2010), `http://dl.acm.org/citation.cfm?id=1924976.1924987`

9. INVEA-TECH: FlowMon Exporter – Community Program (Apr 2013), `http://www.invea-tech.com`

10. Lesk, M.E., Schmidt, E.: Lex – a Lexical Analyzer Generator. Tech. rep., Bell Laboratories (1975), Computing Science Technical Report No. 39

11. Levine, J., John, L.: Flex & Bison. O'Reilly Media, Inc., 1st edn. (2009)

12. Mahanti, A., Williamson, C., Carlsson, N., Arlitt, M., Mahanti, A.: Characterizing the file hosting ecosystem: A view from the edge. Perform. Eval. 68(11), 1085–1102 (Nov 2011), `http://dx.doi.org/10.1016/j.peva.2011.07.016`

13. McNaughton, R., Yamada, H.: Regular Expressions and State Graphs for Automata. Electronic Computers, IRE Transactions on EC-9(1), 39–47 (1960)

14. Open Information Security Foundation: Suricata – network IDS, IPS and network security monitoring engine (Apr 2013), `http://www.suricata-ids.org`

15. Pang, R., Paxson, V., Sommer, R., Peterson, L.: binpac: A yacc for Writing Application Protocol Parsers. In: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement. pp. 289–300. IMC '06, ACM, New York, NY, USA (2006), `http://doi.acm.org/10.1145/1177080.1177119`

16. Paxson, V.: Bro: a system for detecting network intruders in real-time. Comput. Netw. 31(23-24), 2435–2463 (Dec 1999), `http://dx.doi.org/10.1016/S1389-1286(99)00112-7`

17. Qualys, Inc.: LibHTP – security-aware parser for the HTTP protocol (Apr 2013), `http://github.com/ironbee/libhtp`

18. Roesch, M.: Snort - Lightweight Intrusion Detection for Networks. In: Proceedings of the 13th USENIX conference on System administration. pp. 229–238. LISA '99, USENIX Association, Berkeley, CA, USA (1999), `http://dl.acm.org/citation.cfm?id=1039834.1039864`

19. Schneider, F., Agarwal, S., Alpcan, T., Feldmann, A.: The new web: characterizing AJAX traffic. In: Proceedings of the 9th international conference on Passive and active network measurement. pp. 31–40. PAM'08, Springer-Verlag, Berlin, Heidelberg (2008), `http://dl.acm.org/citation.cfm?id=1791949.1791955`

20. Šíma T., Velan P., Čeleda P.: FlowMon - Plugins for HTTP Monitoring (Apr 2013), `http://dior.ics.muni.cz/~velan/flowmon-input-http/`

21. Torres, L., Magana, E., Izal, M., Morato, D.: Identifying sessions to websites as an aggregation of related flows. In: Telecommunications Network Strategy and Planning Symposium (NETWORKS), 2012 XVth International. pp. 1–6 (2012)

22. Torres, L.M., Magana, E., Izal, M., Morato, D.: Strategies for automatic labelling of web traffic traces. 37th Annual IEEE Conference on Local Computer Networks 0, 196–199 (2012)