Libprotoident: Traffic Classification Using Lightweight Packet Inspection

Article · January 2012

CITATIONS
57

READS
2,654

2 authors:

Shane Alcock
University of Waikato
12 PUBLICATIONS **275** CITATIONS
SEE PROFILE

Richard Nelson
University of Waikato
29 PUBLICATIONS **405** CITATIONS
SEE PROFILE

Libprotoident: Traffic Classification Using Lightweight Packet Inspection

Shane Alcock
University of Waikato
Hamilton, New Zealand
salcock@cs.waikato.ac.nz

Richard Nelson
University of Waikato
Hamilton, New Zealand
richardn@cs.waikato.ac.nz

***Abstract*—At present, accurate traffic classification usually requires the use of deep packet inspection to analyse packet payload. This requires significant CPU and memory resources and are invasive of network user privacy. In this paper, we propose an alternative traffic classification approach that is lightweight and only examines the first four bytes of packet payload observed in each direction. We have implemented as an open-source library called libprotoident, which we evaluate by comparing its performance against existing traffic classifiers that use deep packet inspection. Our results show that our approach offers comparable (if not better) accuracy than tools that have access to full packet payload, yet requires less processing resources and is more acceptable, from a privacy standpoint, to network operators and users.**

I. Introduction

Accurate and fast classification of network traffic according to the application type continues to be a difficult problem to solve. The current approach is to utilise solutions that are based on deep packet inspection (DPI) techniques which search the packet content for known application signatures. While this approach is regarded as accurate, it is computationally expensive and introduces concerns regarding user privacy because of the need to access the full payload of all packets. Also, DPI is not practical for offline analysis, which is the approach typically employed by researchers, as the privacy concerns discourage network administrators sharing their data with independant parties and the size of the packet traces makes long-term storage and sharing of full payload packet traces infeasible.

In this paper, we describe a novel approach for traffic classification that only requires four bytes of payload to be retained for each packet, which we refer to as "Lightweight Packet Inspection" (LPI). This approach alleviates both the storage and privacy concerns resulting from the full payload packet captures required by the DPI approach. We have implemented our apprach as an open-source software library, called *libprotoident*, and use the library to demonstrate that LPI is faster and less memory-intensive compared with existing DPI solutions. Furthermore, we shall also show that libprotoident offers comparable accuracy to contemporary DPI tools, despite having access to much less information.

The remainder of the paper is laid out as follows. In the next section, we discuss existing DPI techniques in further detail and discuss the origins of LPI. In Section III, we describe the libprotoident implementation. Libprotoident is evaluated in comparison to several DPI solutions in Section IV, where we examine the speed, memory usage and accuracy of our technique. We conclude the paper in Section V.

II. Background and Related Work

Traditionally, traffic classification techniques fall into one of three categories: port-based (determining the application based on the port numbers used by the endpoints), payload-based (examining the packet payload and matching the protocol based on known application signatures) and statistical (using measurable properties of traffic such as packet size or inter-arrival time to recognise application behaviour). Port-based techniques are widely regarded as insufficient when analysing contemporary traffic as many applications (particularly peer-to-peer file sharing programs) do not use a well-defined port [1]. Some users even configure their programs to use the well-known port for another application to avoid port-based traffic shaping or filtering mechanisms. By contrast, statistical methods, such as [2] and [3], have shown some promise [4] but are not yet regarded as sufficiently reliable for widespread use.

As a result, most traffic classifiers rely on payload-based approaches using deep packet inspection. To do this, the entire contents of each packet must be made available to the traffic classifier. DPI solutions range from open-source software such as OpenDPI [5] and L7 Filter [6] through to proprietary systems developed and sold by companies specialising in traffic management.

Depending on the quality of the application signatures, a DPI approach can produce accurate traffic classifications but there are also several drawbacks: firstly, capturing and processing the entire packet elevates the processing requirements compared to port-based or statistical methods which only examine the packet headers. This impacts on both the affordability and scalability of DPI solutions. Second, the examined payload will often contain sensitive user data that may compromise the privacy of the network users. Finally, DPI is not a viable traffic classification approach for network researchers who typically rely on packet traces stored on disk (such as ourselves). Full payload capture requires huge quantities of disk space and the aforementioned privacy concerns are even greater when

TABLE I
EVALUATED CLASSIFICATION TECHNIQUES

Name	Version	Approach	Tracks IPs?	License
Libprotoident	2.0.2	LPI	No	GPL
PACE	1.31	DPI	Yes	Commercial
OpenDPI	1.2.0	DPI	Yes	LGPL
L7 Filter (TIE)	1.1	DPI	No	GPL
Nmap	5.51	Port-Based	No	GPL

allowing an independent party to access potentially sensitive user data.

Instead, we propose the use of "Lightweight Packet Inspection", where only a maximum of four bytes of payload are examined for each packet. As a result, less processing power is required to classify traffic, trace storage and subsequent analysis becomes feasible and the user privacy concerns, while not eliminated, are greatly reduced. The classification approach that we propose is still primarily payload-based, but it also employs aspects of both port-based and statistical techniques to improve the accuracy.

The selection of four bytes of payload for our technique is based on two observations. Firstly, [7] demonstrated that almost all application signatures start and finish within the first 32 bytes of payload, indicating that a lightweight approach using only a small portion of payload could be viable. Secondly, our experiences in negotiating the installation of passive monitors into both academic and commercial networks showed that network operators were willing to accept the capture of the first four bytes of application payload for research purposes, provided other anonymisation techniques were employed (such as IP address sanitisation) prior to any public release of the data. The operators were satisfied that it was highly unlikely that sensitive data could be determined using only four bytes of packet payload. As a result, many of the packet header trace sets described on the WITS site [8] include four bytes of application payload per packet.

The concept of lightweight payload inspection is not new: [9] describes the traffic classification approach of NetPDL [10] as "lightweight". However, NetPDL is still a DPI approach; the lightweight aspect is that it only inspects the first packet for each session. PortLoad [7] is much closer to our approach in that it utilises only a limited amount of packet payload for traffic classification to avoid the privacy and scalability concerns we noted earlier. The authors demonstrated that a PortLoad instance using only the first 32 bytes of packet payload for each direction achieved 97% byte accuracy when compared against L7 Filter. However, we believe that 32 bytes of payload would still be problematic from a privacy perspective.

III. IMPLEMENTATION

We have implemented our approach as a software library, called libprotoident, with a simple C programming API consisting of only eight functions. The user is responsible for reading packets from the capture source using libtrace [11] and assigning those packets to bidirectional flows (henceforth referred to as *biflows*). Each packet must be passed into libprotoident using the `lpi_update_data` function which will extract any information necessary for traffic classification from the packet. For each biflow, libprotoident records the first four bytes of payload observed in each direction, the port numbers used by both endpoints, the amount of payload present in the first payload-bearing packet for each direction and the IP addresses of both endpoints.

Segment reordering can be a major problem: the first payload-bearing packet that is observed in a capture is not necessarily the first payload-bearing packet transmitted by the sending endpoint. We resolve this for TCP biflows by recording the sequence number of the SYN packet sent in each direction, so that we can correctly identify the first segment by looking for a sequence number exactly one higher than the SYN sequence number.

The classification of a biflow occurs when the `lpi_guess_protocol` function is called, which passes the recorded information for that biflow to the rule matching function for each supported application protocol until a match is found. Libprotoident currently supports 186 unique TCP and UDP application protocols, each of which is implemented as a separate module with a priority value ranging from 1 (very high priority) to 255 (very low priority). The priority determines the order in which the rule matching functions are run and is based on both our confidence in the rules for that protocol and the popularity of the application. For example, HTTP is given a slightly higher priority than other protocols with equally strong rules to try and minimise the number of rule matching functions that must be run to achieve a successful match.

A rule matching function returns true if the biflow meets the requirements for the application protocol and false otherwise. The function will consist of one or more rules that must be met by the biflow to result in a successful match. There are four types of rule that a libprotoident protocol module can use to identify the application protocol for a given biflow:

Payload Matches: This is the most common type of rule in libprotoident, whereby the four bytes of recorded payload is compared against a known signature for the protocol. For example, the BitTorrent rule matching function simply looks for the character pattern '0x13', 'B', 'i', 't'. A rule may contain specific characters for all four bytes or may include a special "any" character which will match anything. In some instances a payload matching rule will enforce request/response behaviour, i.e. the request pattern must be observed in one direction and the response pattern must be observed in the other, but two requests or two responses are not allowed. Because the payload matching is limited to 4 bytes, the comparisons can be done by treating both the payload and the pattern as 32-bit integers and comparing them directly.

Payload Size: Many protocols do not have a clearly identifiable pattern in the first four bytes of payload or have an ambiguous payload pattern, e.g. several different game protocols all fill the first four bytes with '0xff' characters. However, it may still be possible to identify these protocols

based on the size of the first payload-bearing packet. For example, one of our rules for matching Skype traffic requires the initial payload in one direction to be exactly 11 bytes. Payload size rules are often integrated with payload matching rules, as many application protocols include a length field in the protocol header. The payload size recorded by libprotoident can be used to determine if the payload contains a correct value for the length field.

Port Number: While port numbers alone are not an accurate means for classifying all traffic, they can still be useful to augment a rule matching function that would otherwise be weak. For instance, the MySQL module in libprotoident has a payload matching rule that accepts any biflows where the fourth byte observed in one direction is '0x00' and the other direction observed no payload at all. This rule would be prone to producing numerous false positives, so it was extended to only apply to flows where one of the endpoints is using TCP port 3306.

IP Matching: This is a special case of payload matching, whereby the four bytes of payload match the IP address of one of the biflow endpoints. This type of rule is only used in libprotoident to match Gnutella UDP Out of Band messages. One weakness of this type of rule is that it will not work if the addresses within the IP header have been sanitised by the packet capture process which is common in publicly released trace sets.

Some DPI-based traffic classification techniques also require state to be maintained for each observed IP address. Traffic on the same IP and port as a previously identified biflow can then be designated to be the same application, even if it does not explicitly match a known pattern or rule. However, IP tracking is memory-intensive, especially on busy links where the number of unique hosts may be large. Because of this, libprotoident does **not** maintain any state for individual IP addresses and instead treats each biflow independently. It is still possible to develop a tool based on libprotoident that does maintain IP state, but the IP tracking must be implemented i separately by the libprotoident user.

IV. Evaluation

To evaluate the performance of libprotoident, we compared our library against three separate DPI approaches and a port-based approach, which are summarised in Table I. We had also hoped to include PortLoad [7] in our evaluation experiments, but the PortLoad software is not freely available and our inquiries regarding access for research purposes were not answered.

PACE [12] is a protocol and application identification engine developed and sold commercially by ipoque, who provided us with a copy of the underlying traffic classification library under a research license for the purpose of this evaluation. OpenDPI [5] is the open-source version of the ipoque engine, which we believe uses the rules and algorithms from an older edition of PACE. L7 Filter [6] is a DPI-based classifier for Linux Netfilter that matches traffic based on patterns defined using regular expressions. Finally, the Nmap technique uses the port to application mappings included with the Nmap tool [13] to classify traffic. To do this, we developed our own custom software to parse the port mapping file and match biflows to an application protocol based on the ports used by the flow endpoints.

TABLE II
Traces Used for Evaluation

Name	Date	Duration	Traffic	Biflows
Auckland	2010/03/22	4 hrs	46.9 GB	5.72 M
ISP	2011/06/13	4 hrs	108.0 GB	7.02 M

All of the classifiers, except for L7 Filter, were integrated into a single program that read packets from an input source, maintained a table of active biflows and requested a traffic classification when the biflow ended or expired. The subscriber hashmap implementation provided by the PACE library was used for IP tracking for those classifiers that required it and the packets were read and processed using libtrace [11]. The program was designed such that a single classifier could be run alone, i.e. for performance tests, or all classifiers could be run concurrently.

Because L7 Filter is not implemented as a software library that could be easily integrated with our evaluation program, we instead used the Traffic Identification Engine (TIE) [14] to evaluate L7 Filter. TIE implements an L7 Filter plugin which replicates the L7 Filter classification approach. TIE also provided all the packet processing and biflow table maintenance for the L7 Filter evaluation.

A. Datasets

Two datasets were used to evaluate our software, which are described in Table II. The datasets were filtered to only contain flows that both started and ended within the capture period to avoid disadvantaging any technique that relies on seeing either the start or the end of the biflow. The traffic and biflow counts given in Table II represent the traffic remaining after the filtering was performed.

The Auckland dataset was a full payload packet capture taken at the University of Auckland in March 2010. The Auckland capture was written to disk on the passive monitor, making it ideal for repeated analysis (such as measuring resource usage). However, the University policy on Internet usage may mean that the applications present in the dataset are biased in favour of protocols that are relatively easy to identify, such as HTTP and SMTP.

The ISP dataset was a full payload packet capture taken from within the core network of a New Zealand ISP which should produce a greater variety of application protocols than the Auckland data. Due to our monitoring agreement with the ISP, we could not write the full payload capture to disk. Our comparison experiments were therefore run live using all of the classification techniques at the same time. To reduce the load on the passive monitor, we limited our analysis to only residential DSL subscribers.

TABLE III
ACCURACY OF THE OPEN SOURCE CLASSIFIERS (AUCKLAND)

Name	Libprotoident	OpenDPI	Nmap	L7 Filter
Matched	57.21%	82.73%	51.61%	57.15%
Different	0.58%	0.65%	43.94%	5.67%
Unknown	0.31%	0.89%	4.45%	6.78%
HTTP Subclassing	34.66%	15.73%	0.00%	30.37%
SSL Subclassing	7.24%	0.00%	0.00%	0.00%

TABLE IV
ACCURACY OF THE OPEN SOURCE CLASSIFIERS (ISP)

Name	Libprotoident	OpenDPI	Nmap	L7 Filter
Matched	47.62%	75.68%	31.03%	46.22%
Different	6.77%	2.76%	45.19%	6.22%
Unknown	7.08%	20.66%	23.79%	15.80%
HTTP Subclassing	33.15%	0.91%	0.00%	31.76%
SSL Subclassing	5.39%	0.00%	0.00%	0.00%

B. Accuracy

To begin, we compared the relative accuracy of the various classification techniques using both the Auckland and ISP datasets. For this experiment, we used the classifications produced by PACE as the measure of ground truth, ignoring all biflows that PACE was unable to identify. In the Auckland dataset, PACE failed to identify 1.4 million biflows accounting for 1.95 GB of traffic, leaving 4.3 million biflows and 45 GB traffic available for the evaluation analysis. In the ISP dataset, 2.6 million biflows representing 5.6 GB of traffic were ignored, resulting in a dataset containing 4.4 million biflows constituting 102 GB of traffic.

The accuracy of each of the remaining techniques is shown in Tables III and IV. The "Matched" category describes the proportion of traffic where the classification directly matched the result reported by PACE. "Different" describes traffic where the classifier managed to report a protocol but the protocol did not match the protocol reported by PACE. "Unknown" refers to traffic where the classifier was unable to suggest any protocol For the purposes of this evaluation, we treat both the "Different" and "Unknown" categories as failure cases.

Furthermore, many applications use HTTP to transport data, e.g. streaming media, and some DPI techniques, including PACE, examine the User-Agent and Content-Type fields inside the HTTP header to further classify HTTP biflows according to the type of media being transported. Some examples of HTTP subclasses reported by PACE include flashvideo, mpeg, windowsmedia and quicktime. We argue that a classifier that reports "HTTP" for those flows is not incorrect, as HTTP is still being used; rather, PACE is simply able to provide additional detail. To distinguish these cases from instances where the classifiers clearly disagree, we have included a separate category called "HTTP Subclassing". Similarly, libprotoident sub-classifies some SSL traffic as either HTTPS or IMAPS based on the port numbers involved. PACE does not make such distinctions, so we have introduced an "SSL Subclassing" category for such cases.

Libprotoident outperformed the other three approaches when analysing the Auckland dataset, despite having access to only a limited amount of payload. Libprotoident failed to identify correctly less than 1% of the traffic classified by PACE in the Auckland dataset, compared with 1.5% for OpenDPI, 12.3% for L7 Filter and 48% for the port-based technique. OpenDPI achieved the greatest proportion of direct matches, due to correctly subclassing much of the HTTP traffic. Nmap achieved a surprisingly high direct match rate using the Auckland dataset, which we believe is due to the lack of peer-to-peer traffic in the Auckland dataset (according to PACE, BitTorrent, EDonkey and Gnutella combined account for less than 3% of all traffic).

When examining the ISP dataset, the failure rate for libprotoident was higher at 13.7%, but still performed better than the other techniques as OpenDPI, L7 Filter and Nmap had failure rates of 23.3%, 22% and 68.9% respectively. 86% of the traffic in the "Different" category for libprotoident was classifed as the UUSee protocol by PACE, whereas our software reported either Skype or UDP BitTorrent for the same traffic. Most of the "Unknown" traffic reported by libprotoident appeared to be encrypted TCP BitTorrent biflows (76%), which we suspect PACE was able to match as a result of observing previous BitTorrent activity between the endpoints involved.

In both datasets, PACE is able to subclassify a significant quantity of HTTP traffic that libprotoident simply reports as HTTP. This is one weakness of the four-byte technique used by libprotoident: the lack of payload makes it impossible to analyse the HTTP header contents and subclassify HTTP traffic accordingly. If that level of traffic classification is required, then the lightweight approach we propose in this paper will not be sufficient and a DPI approach will be necessary.

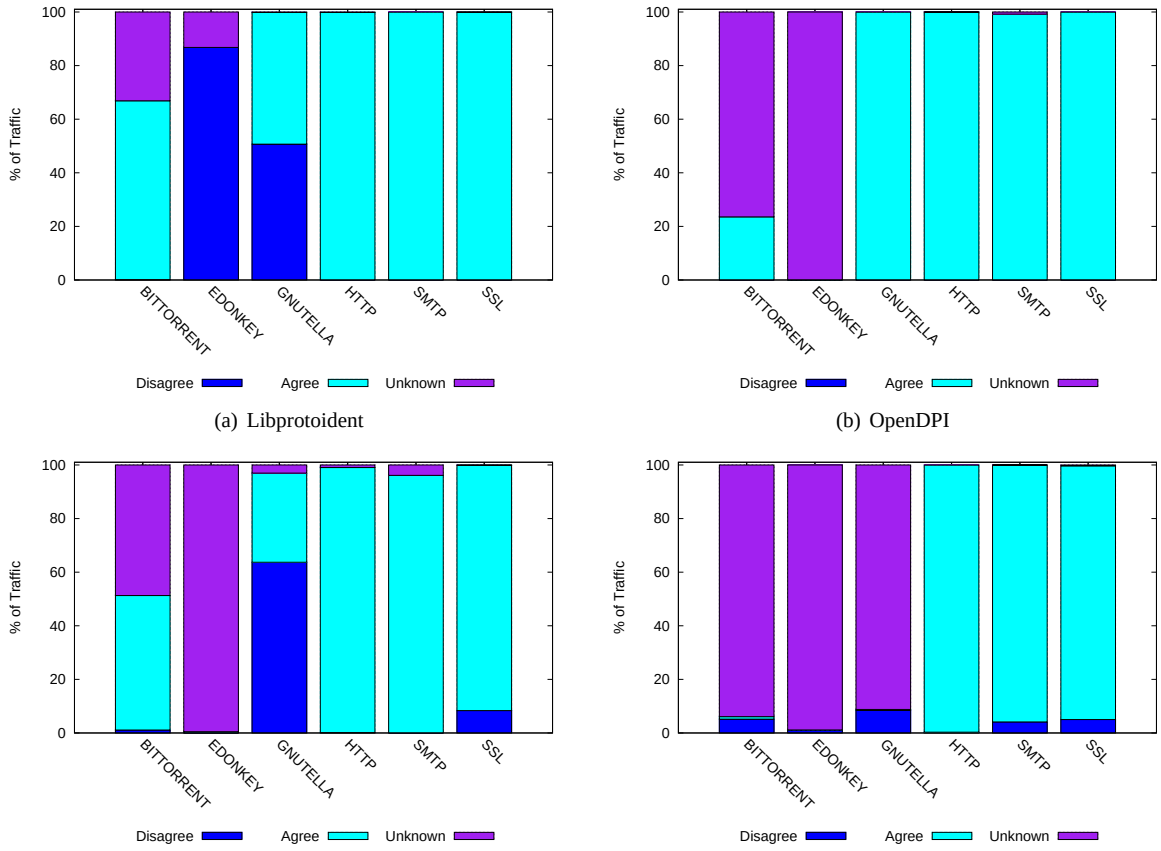
TABLE V
ADDITIONAL TRAFFIC CLASSIFED BY LIBPROTOIDENT

	Auckland		ISP	
Rank	Protocol	Bytes	Protocol	Bytes
1	ESP over UDP	90 MB	Skype	203 MB
2	HTTP	20 MB	RTMP	125 MB
3	BitTorrent UDP	11 MB	Xbox Live	122 MB
4	Razor	7.6 MB	BitTorrent UDP	94 MB
5	Garena	7.0 MB	HTTP	35 MB

C. Unknown Traffic

In addition, we examined biflows that PACE reported as unknown but were classified by libprotoident. The total amount of additional traffic classified by libprotoident and the top five contributing protocols are shown in Table V. Overall, the amount of traffic that libprotoident classified but PACE did not was almost negligible: 178 MB in the Auckland dataset and 755 MB in the ISP dataset, which was less than one percent of the traffic in each dataset.

We note that PACE appeared to miss some HTTP traffic in both datasets. Examining the source code for OpenDPI (which we assume uses similar rules for HTTP as PACE), we found that OpenDPI would fail to identify HTTP flows where the GET command and the "Host:" field were not in the same TCP segment. This can occur when the length of the GET



(a) Libprotoident (b) OpenDPI

(c) L7 Filter (d) Nmap

Fig. 1. Comparing accuracy when classifying specific protocols in the ISP dataset, using PACE as ground truth.

object exceeds the maximum segment size, e.g. if the requested URL is very long. Also, some web clients deliberately use separate writes for the GET command and the "Host:" field, resulting in two distinct objects. If PACE has the same problem as OpenDPI, that may explain the failure to identify some HTTP traffic correctly.

D. Protocols

We also measured the accuracy of each approach when identifying specific application protocols. The ISP dataset was used for this experiment, due to the greater likelihood of peer-to-peer applications being used. Again, we used the PACE classifications as the ground truth and selected six protocols that may be of interest to the users of traffic classifiers, including BitTorrent, HTTP and SSL. The results are shown in Figure 1.

Libprotoident was the best of the four techniques at identifying BitTorrent, successfully identifying 65% of the BitTorrent traffic reported by PACE, and achieved near 100% accuracy when classifying HTTP, SMTP and SSL traffic. However, our library disagreed with the classification of much of the EDonkey and Gnutella traffic. It should be noted that neither EDonkey nor Gnutella contributed a significant quantity of traffic in the ISP dataset; PACE reported 25 MB of EDonkey traffic and 50 MB of Gnutella. By contrast, there were 19 GB of BitTorrent traffic observed in the ISP dataset.

TABLE VI
USER-MODE CPU SECONDS USED BY EACH CLASSIFIER

Tool	Minimum	Mean	Maximum	Ratio to Nmap
Nmap	426.36 s	430.67 s	434.68 s	1.00
Libprotoident	478.63 s	484.45 s	489.11 s	1.12
OpenDPI	563.83 s	568.11 s	579.20 s	1.32
PACE	678.90 s	684.30 s	688.15 s	1.59
L7 Filter	1540.07 s	1550.16 s	1558.39 s	3.60

The traffic identified as EDonkey by PACE was mostly classified as UDP BitTorrent by libprotoident. This was because the recorded payload matched the header format for the Distributed Hash Table used by the Vuze BitTorrent client [15]. We believe this to be the correct classification and note that none of the other classification techniques agree with PACE in this instance. The Gnutella disagreement is primarily due to the Gnutella traffic being encrypted using SSL. IP tracking allows PACE to match the encrypted traffic as SSL based on previous Gnutella activity by that host, whereas libprotoident can only recognise the traffic as SSL.

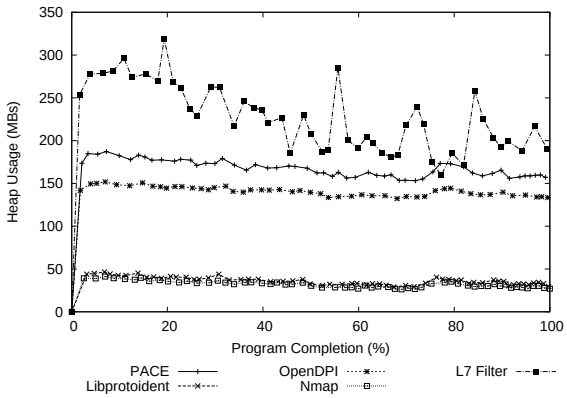


Fig. 2. Heap usage of the evaluated traffic classifiers.

E. Speed

We measured the CPU usage for each of the evaluated techniques when run over the Auckland dataset. Each technique was run against the dataset ten times and the total number of CPU-seconds used directly by the process (i.e. in user mode) was measured using the Unix time tool. The trace files were decompressed using a separate process and the uncompressed packet data was piped into the traffic classifier via stdin.

The mean, maximum and variance of the CPU measurements are shown in Table VI. The final column reports the ratio of mean CPU-seconds required by the given approach to CPU-seconds required by the port-based Nmap method which does not perform any payload analysis at all. The results show that libprotoident is faster than all of the DPI approaches, which is not surprising, and requires only 12.5% additional CPU time than the port-based technique.

We note that the commercial ipoque tool is slower than the open source equivalent, suggesting that the improved accuracy of PACE does come at a performance cost. L7 Filter proved to be the slowest of the tools by a significant margin, possibly due to the use of regular expressions to match payload patterns which are more computationally expensive than fixed string payload matches. Further investigation showed that a TIE instance using the L7 Filter plugin executed 5.68 times as many instructions as PACE, which executed the next highest number of instructions.

F. Memory Usage

Finally, we examined the memory footprint of each of the tools when processing the Auckland dataset. We used the *massif* memory profiling tool for this purpose [16], which periodically reports heap utilisation. As with the CPU measurements, the trace files were decompressed using a separate process. The memory usage of each tool over the course of the traffic classification process is depicted in Figure 2. The X-axis describes the percentage of the program that had been completed at the time that the snapshot of heap usage had been taken.

The graph shows that libprotoident uses only marginally more memory than a port-based approach, suggesting the additional memory requirements of our approach are minimal. By comparison, OpenDPI and PACE use over three times as much memory as libprotoident. We suspect much of this additional memory usage can be attributed to IP tracking although both tools may also record more state per-flow than libprotoident requires. Finally, L7 Filter once again proves to be the most resource-intensive of the tools, peaking at over 300 MB of heap usage (libprotoident uses less than 50 MB of heap memory throughout).

V. Conclusion

In this paper, we have described a new traffic classification approach that requires only four bytes of packet payload to be retained for each packet, called Lightweight Packet Inspection. Our approach resolves or alleviates many of the drawbacks of Deep Packet Inspection and allows both researchers and network operators to employ accurate traffic classification in scenarios where DPI would be inappropriate or too expensive to employ. We have implemented our approach as an open-source library, libprotoident, which supports over 180 unique application protocols.

We have evaluated the performance of libprotoident in comparison with existing DPI solutions, including both open-source and commercial software, and found that libprotoident is more accurate at classifying traffic than existing open-source DPI software, despite the relative lack of information. Furthermore, libprotoident is faster and requires significantly less memory than any of the examined DPI software. These results vindicate our approach and prove that full payload capture is not necessary to perform accurate payload-based traffic classification.

Libprotoident itself is an ongoing project with new rules and protocol support added on a frequent basis. Future work also includes exploring parallelising the matching of rules with the same priority to further improve performance. Another possible extension is to investigate automated detection and learning of common payload patterns and sizes to ensure that new protocols can be quickly detected and appropriate rule matching functions developed.

VI. Acknowledgments

We would like to thank Klaus Mochalski and Ralf Hoffman at ipoque for providing us with access to PACE, which gave us a genuine benchmark to compare our software against. We also acknowledge the help of Alberto Dainotti, who helped us with TIE.

References

[1] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos, "Is P2P Dying or Just Hiding?" 2004.

[2] Marcell Pernyi and Trang Dinh Dang and A. Gefferth and S. Molnr, "Identification and Analysis of Peer-to-Peer Traffic," *Journal of Communications*, vol. 1, no. 7, pp. 36–46, 2006.

[3] M. Crotti and M. Dusi and F. Gringoli and L. Salgarelli, "Traffic Classification through Simple Statistical Fingerprinting," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 1, pp. 5–16, 2007.

[4] M. Pietrzyk and J-L. Costeux and G. Urvoy-Keller and T. En-Najjary, "Challenging Statistical Classification for Operational Usage: the ADSL Case," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, 2009, pp. 122–135.
[5] "OpenDPI," http://www.opendpi.org/.
[6] Clear Foundation, "l7-filter," http://l7-filter.clearfoundation.com/.
[7] G. Aceto, A. Dainotti, W. de Donato, and A. Pescap, "PortLoad: Taking the Best of Two Worlds in Traffic Classification," in *IEEE INFOCOM 2010 - WIP Track*, 2010.
[8] WAND Network Research Group, "WITS: Waikato Internet Traffic Storage," http://www.wand.net.nz/wits/index.php.
[9] Fulvio Risso and Mario Baldi and Olivier Morandi and Andrea Baldini and Pere Monclus, "Lightweight, Payload-Based Traffic Classification: An Experimental Evaluation," in *ICC'08*, 2008, pp. 5869–5875.
[10] Computer Networks Group at Politecnico di Torino, "The NetBee Library," http://www.nbee.org/.
[11] WAND Network Research Group, "Libtrace," http://research.wand.net.nz/software/libtrace.php.
[12] ipoque, "Protocol and Application Classification Engine (PACE)," http://www.ipoque.com/products/pace-application-classification.
[13] "Nmap," http://nmap.org/.
[14] A. Dainotti, W. Donato, and A. Pescapé, "Tie: A community-oriented traffic classification platform," in *Proceedings of the First International Workshop on Traffic Monitoring and Analysis*, 2009, pp. 64–74.
[15] Vuze, "Distributed hash table," http://wiki.vuze.com/w/Distributed_hash_table.
[16] Valgrind Developers, "Valgrind User Manual: Massif," http://valgrind.org/docs/manual/ms-manual.html.