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 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 our approach 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 and requires less processing resources.

Categories and Subject Descriptors

C.2.2 [**Computer Communications Networks**]: Network Protocols

General Terms

Measurement

Keywords

Deep Packet Inspection, Application Classification, Open Source, Privacy

1. 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 can be accurate, it is computationally expensive and requires access to the full payload of all packets, which may be problematic from a user privacy perspective. Also, DPI is not practical for offline analysis, which is typically employed by researchers, as the privacy concerns discourage network administrators sharing their data with independent parties and full payload packet traces are too large to be able to store and share in a feasible fashion.

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 alleviates both the storage and privacy concerns associated with DPI. We have implemented our approach 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 also show that libprotoident offers comparable accuracy to contemporary DPI tools, despite the decreased amount of captured payload.

2. BACKGROUND AND RELATED WORK

Broadly speaking, traffic classification techniques fall into one of three categories: port-based, payload-based and statistical. 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 also 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 which examine the packet or object sizes, such as [2] and [3], have shown some promise [4] but are not yet regarded as sufficiently reliable for widespread use.

Most traffic classifiers rely on payload-based methods, particularly deep packet inspection (DPI). DPI requires that 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 can limit 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. Full payload capture requires huge quantities of disk space and the aforementioned privacy concerns are even greater when allowing an independent party to store and 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 elements of both port-based and statistical techniques to improve 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 user privacy could be compromised using only four bytes of packet payload.

The concept of lightweight payload inspection is not new: [8] describes the traffic classification approach of NetPDL [9] as "lightweight". However, NetPDL is still a DPI approach but is claimed to be lightweight because it only inspects the first packet for each session instead of every packet. 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 (an open-source DPI tool).

Listing 1: The update data function (pseudocode).

```
int lpi_update_data(packet, LPIdata, dir) {
    int psize = get_payload_length(packet);
    char *payload = get_payload_start(packet);

    /* If we've already seen payload for this direction, ignore */
    if (LPIdata->plen[dir] != 0)
        return 0;

    LPIdata->transport = get_transport(packet);

    /* For TCP packets, we need to be careful about reordering */
    if (get_transport(packet) == TCP) {
        if (tcp_syn(packet)) {
            LPIdata->expected_seq = tcp_seq(packet) + 1;
        }
        if (tcp_seq(packet) != LPIdata->expected_seq)
            return 0;
    }

    /* Record the port numbers used */
    if (LPIdata->port[dir] == 0) {
        LPIdata->port[dir] = get_source_port(packet);
        LPIdata->port[other_dir] = get_dest_port(packet);
    }

    /* Only interested in payload-bearing packets */
    if (psize == 0 || payload == NULL)
        return 0;

    /* Grab the first four bytes of payload only */
    uint32_t four_bytes = *(uint32_t *)payload;

    /* Zero any extra bytes in the uint32_t */
    if (psize < 4) {
        four_bytes = byteswap(four_bytes) >> (8 * (4 - psize));
        four_bytes = byteswap(four_bytes << (8 * (4 - psize)));
    }

    /* Record the payload and length for this direction */
    LPIdata->payload[dir] = four_bytes;
    LPIdata->plen[dir] = psize;

    /* Grab the IP addresses too */
    LPIdata->ips[dir] = get_source_ip(packet);
    LPIdata->ips[otherdir] = get_dest_ip(packet);

    return 1;
}
```

3. IMPLEMENTATION

We have implemented our approach as a software library, called libprotoident, with a simple C programming API. When using libprotoident, the user must read packets from the capture source using libtrace [10] and assign those packets to bidirectional flow records (henceforth referred to as *biflows*). The user is entirely responsible for the management and expiry of flow records.

In addition, each biflow must have an instance of an LPI data structure associated with it, where libprotoident stores the information required for the traffic classification. Each packet read must be passed into the libprotoident update data function, along with the LPI data structure for the packet's biflow and the direction the packet was travelling, which extracts the needed information from the packet. The direction value may be either zero or one; all packets sent by one endpoint must use one of the direction values while all packets sent by the other endpoint must use the other.

The update function (shown in Listing 1 records the first four bytes of payload observed in each direction, the amount of payload present in the first payload-bearing packet for each direction and the ports and IP addresses used by the two endpoints. The update function also compensates for any possible TCP segment re-ordering by recording the expected sequence number of the first payload-bearing segment in each direction. No effort is made to reorder UDP segments due to the lack of ordering information available.

The classification of a biflow occurs when the match protocol function is called, which calls the rule matching function for each supported application protocol until a match is found. Libprotoident currently supports over 200 unique TCP and UDP application protocols [11], each of which is implemented as an independent module. The modules are registered when libprotoident is initialised. Each protocol module has a priority value ranging from 1 (very high priority) to 255 (very low priority). The priority determines the order in which the protocol modules are run and is based on both our confidence in the rule matching function and the popularity of the application. The algorithm used for this process is described in Listing 2.

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 classify 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. A rule may contain specific characters for all four bytes or may include a special "any" wildcard character. 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. 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, removing the need for complicated bit field analysis such as that employed by [7].

Payload Size: Many protocols do not have a clearly identifiable pattern in the first four bytes of payload or have an ambiguous payload pattern. However, it may still be possible to identify these protocols based on the size of the first payload-bearing packet. Payload size rules are often inte-

Table 1: 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

Listing 2: The match protocol function.

```
lpi_module_t *run_modules(LPIdata, modules) {
    for (priority = 1; priority <= 255; priority++) {
        foreach mod in modules[priority] {
            if (mod->match(LPIdata))
                return mod;
        }
    }
    return NULL;
}

lpi_module_t *lpi_match_protocol(LPIdata) {
    lpi_module_t *p = NULL;
    switch(LPIdata->transport) {
        case ICMP:
            p = lpi_icmp_module; break;
        case TCP:
            p = run_modules(TCP_modules, LPIdata);
            if (p == NULL)
                p = lpi_unknown_tcp;
            break;
        case UDP:
            p = run_modules(UDP_modules, LPIdata);
            if (p == NULL)
                p = lpi_unknown_udp;
            break;
        default:
            p = lpi_unsupported;
    }
    return p;
}
```

Listing 3: The PPStream rule matching functions.

```
static bool ppstream_payload(payload, len) {

    uint16_t len_field = 0;
    uint32_t swap = byteswap(payload);

    if (len == 0)
        return true;

    if (!MATCH(payload, ANY, ANY, 0x43, 0x00))
        return false;
    len_field = byteswap((uint16_t)(swap >> 16));

    /* Some versions of PPStream use length, others use
     * length - 4 */
    if (len_field == len)
        return true;
    if (len_field == len - 4)
        return true;
    return false;
}

bool match_ppstream(LPIdata) {

    if (!ppstream_payload(LPIdata->payload[0], LPIdata->plen[0]))
        return false;
    if (!ppstream_payload(LPIdata->payload[1], LPIdata->plen[1]))
        return false;
    return true;
}
```

grated 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 length field has the correct value.

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 or prone to false positives. Port numbers are only be used for applications that have a well-defined port number, such as DNS on port 53.

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.

The rule matching function for the PPStream protocol is shown in Listing 3 which demonstrates how the different types of rule are used in practice. To match PPStream, the payload for both directions must meet certain requirements. Specifically, a payload matching rule is used to enforce that the third and fourth bytes of payload are 0x43 and 0x00 respectively. The wildcard character ANY is used to ignore the values in the first two bytes, as the PPStream protocol uses them as a length field. To check whether the length field is valid, the value of the first two bytes is compared against the recorded payload size. In addition, one-way PPStream flows are supported by returning true if no payload was observed in a given direction. Biflows with no payload in both directions have already been classified by the time this function is called, so there is no chance of them being erroneously reported as PPStream.

Some DPI-based traffic classification techniques also record the IP address and port for each successful match. Subsequent 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, this approach 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 state for each IP and port, but this must be implemented separately.

4. 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 1. 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. PACE has been used to conduct yearly studies of Internet traffic, including [13], and is also used to run the Internet Observatory project [14]. As a result, we decided that PACE would be a suitable representative of the current "state-of-the-art" in the area of traffic classification.

OpenDPI [5] is the open-source version of the ipoque engine which 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 [15] 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.

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 [10]. 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) [16] 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.

4.1 Datasets

Two datasets were used to evaluate our software, which are described in Table 4. The datasets were filtered to only contain flows that both started and ended within the capture period to avoid disadvantaging any technique that relied on seeing either the start or the end of the biflow. The traffic and biflow counts given in Table 4 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.

4.2 Accuracy

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.

Table 2: Evaluated Accuracy (Auckland data)

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 Sub.	34.66%	15.73%	0.00%	30.37%
SSL Sub.	7.24%	0.00%	0.00%	0.00%

Table 3: Evaluated Accuracy (ISP data)

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 Sub.	33.15%	0.91%	0.00%	31.76%
SSL Sub.	5.39%	0.00%	0.00%	0.00%

The accuracy of each of the remaining techniques is shown in Tables 2 and 3. 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 "Different" and "Unknown" 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 sub-classes reported by PACE include flashvideo, mpeg, windowsmedia and quicktime. In both datasets, PACE is able to sub-classify a significant quantity of HTTP traffic that libprotoident simply reports as HTTP. To distinguish these cases from instances where the classifiers clearly disagree, we have included a separate category called "HTTP Sub-classing".

This is one weakness of the four-byte technique used by libprotoident: it is impossible to analyse the HTTP header contents and sub-classify HTTP traffic accordingly. Unfortunately, this cannot be solved by simply capturing a few more bytes as the fields used by DPI techniques are often deep inside the HTTP payload; an extra four or twelve bytes will not help.

A similar case arises with SSL traffic. In this instance, libprotoident is able to sub-classify some SSL traffic as either HTTPS or IMAPS based on port numbers. PACE does not make such distinctions, so we have also introduced an "SSL

Table 4: 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

Table 5: Extra Traffic Classified by Libprotoident

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

Sub-classing" category to these cases where libprotoident provides more detail than PACE.

Libprotoident outperformed the three open source libraries 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 sub-classing 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 (BitTorrent, EDonkey and Gnutella combined account for less than 3% of all traffic, according to PACE).

When examining the ISP dataset, the failure rate for libprotoident was higher at 13.7%, but it 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 classified 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.

Libprotoident was the best of the four techniques at identifying BitTorrent, successfully identifying 65% of the BitTorrent traffic reported by PACE in the ISP dataset, 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. The traffic identified as EDonkey by PACE was mostly classified as UDP BitTorrent by libprotoident. This was because the four bytes of payload matched the header format for the Distributed Hash Table used by the Vuze BitTorrent client [17]. The Gnutella disagreement is primarily due to the Gnutella traffic being encrypted using SSL. IP tracking allowed PACE to classify the encrypted traffic based on previous Gnutella activity by that host, whereas libprotoident can only recognise the traffic as SSL.

However, 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. These quantities are far outweighed by the 19 GB of BitTorrent traffic observed in the ISP dataset.

4.3 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 5. 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 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.

Table 6: User-mode CPU seconds used

Tool	Minimum	Mean	Maximum	Ratio
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

4.4 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 6. 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 was faster than all of the DPI approaches and required only 12.5% additional CPU time than the port-based technique.

We note that the commercial ipoque tool was 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.

4.5 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 [18] to periodically report heap utilisation as each tool was running. As with the CPU measurements, the trace files were decompressed using a separate process. The memory usage of each tool over the course

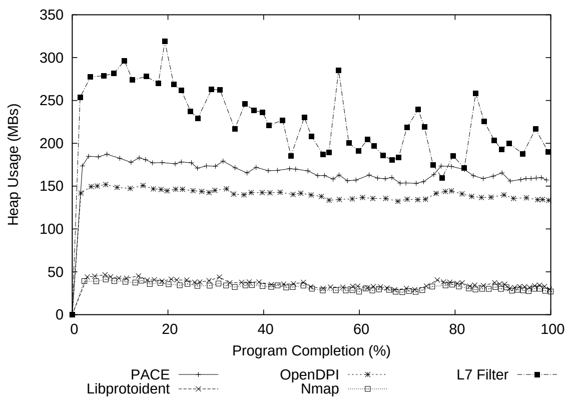

Figure 1: Heap usage of the evaluated classifiers.

of the traffic classification process is depicted in Figure 1. 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 used only marginally more memory than a port-based approach, suggesting the additional memory requirements of our approach are minimal. By comparison, OpenDPI and PACE used 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. Finally, L7 Filter once again proved to be the most resource-intensive of the tools, peaking at over 300 MB of heap usage.

5. CONCLUSION

In this paper, we have described a new traffic classification approach, called Lightweight Packet Inspection, that requires only four bytes of packet payload to be retained for each packet. 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 200 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 strictly 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.

6. ACKNOWLEDGEMENTS

We would like to thank Klaus Mochalski and Ralf Hoffman at ipoque for providing us with access to PACE and Alberto Dainotti for helping us with TIE.

7. AVAILABILITY

The latest version of libprotoident (2.0.3) is available for download at `http://research.wand.net.nz/`.

8. 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 Comput. Commun. Rev.*, 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] 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.

[9] Computer Networks Group at Politecnico di Torino, "The NetBee Library," http://www.nbee.org/.

[10] WAND Network Research Group, "Libtrace," http://research.wand.net.nz/software/libtrace.php.

[11] ——, "Libprotoident: Supported Protocols," http://www.wand.net.nz/trac/libprotoident/wiki/SupportedProtocols.

[12] ipoque, "Protocol and Application Classification Engine (PACE)," http://www.ipoque.com/products/pace-application-classification.

[13] H. Schulze and K. Mochalski, "Internet Study 2008/2009," *Africa*, pp. 1–13, 2009.

[14] "Internet Observatory," http://www.internetobservatory.net/.

[15] "Nmap," http://nmap.org/.

[16] 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.

[17] Vuze, "Distributed hash table," http://wiki.vuze.com/w/Distributed_hash_table.

[18] Valgrind Developers, "Valgrind User Manual: Massif," http://valgrind.org/docs/manual/ms-manual.html.