# nDPI: Open-Source High-Speed Deep Packet Inspection

Luca Deri *†, Maurizio Martinelli*, Alfredo Cardigliano†
IIT/CNR*
ntop†
Pisa, Italy
{luca.deri, maurizio.martinelli}@iit.cnr.it

***Abstract*—Network traffic analysis has been traditionally limited to packet header as the transport protocol and application ports were usually enough to identify the application protocol. With the advent of port-independent, peer-to-peer and encrypted protocols, the task of identifying application protocols has become increasingly challenging, thus paving the way to the creation of tools and libraries for network protocol classification.**

**This paper covers the design and implementation of nDPI, an open-source library for the protocol classification through the analysis of both packet header and payload. nDPI has been validated extensively in various monitoring projects ranging from Linux kernel protocol classification, to analysis of suspicious communications targeting the .it ccTLD.**

***Keywords—Passive protocol classification, deep-packet-inspection, network traffic monitoring.***

## I. INTRODUCTION

In the early days of the Internet, network traffic protocols were bound to a specific protocol and port. For instance the SMTP protocol used TCP and port 25, as well telnet TCP and port 23. This well-know protocol/port association is specified in the /etc/protocols file part of every Unix-based operating system. Over time, the use of static ports has become a problem in particular with the advent of RPC (Remote Procedure Call) and thus specific applications such as rpcbind and portmap have been developed to handle these dynamic mappings. Historically the application ports up to 1024 identified essential system services such as email or remote system login and thus they require super-user privileges, and their port-to-protocol binding has been preserved even today. The remaining ports above 1024, are instead used for user-defined services and they generally use dynamic ports.

The fact that a protocol uses a static port is often not enough to correctly identify the real protocol being used. An example of this practice is TCP/80 used by HTTP. Originally HTTP was used to carry web-related resources such as HTML pages and icons used to decorate web pages (and thus its name HyperText Protocol). Thanks to its native integration in web browsers and flexibility due to the versatile HTTP header where the MIME type could be specified, HTTP is often used to carry non web-related resources. For instance it is not the de-facto protocol for downloading/uploading files, thus replacing the FTP (File Transfer Protocol) that was designed exactly for that purpose. The pervasive HTTP use and its native support on firewall (i.e. they can recognise the protocol header in order to validate it), has made HTTP (and its secure counterpart HTTPS) the ideal choice for developers when creating a new protocol that was supposed to pass-through firewalls without restrictions. Many peer-to-peer protocols or popular applications such as Skype, use HTTP as last resort when they need to pass through a firewall where all other ports were blocked. We have created traffic reports from various networks ranging from academic sites to commercial ISPs, and realised that HTTP is by far the most widely used protocol. This however does not mean that users mostly use it for surfing the web, as this protocol is so pervasive that social networks, maps, and video-streaming services use it extensively. In other words the equation TCP/80 = web, no longer holds.

Characterising network protocols is necessary not just for creating accurate reports of network traffic exchanged on our home network, but also for increasing the overall network security. Modern firewalls are able to combine IP/protocol/port based security with selected protocol inspection in order to validate protocols, in particular those based on UDP such as SNMP (Simple Network Management Protocol) and DNS (Domain Name System). Other VoIP-based protocols such as SIP and H.323 are instead inspected to extract specific information (e.g. the IP and port where voice and video will flow) necessary for the firewall to know what IP/ports to open in order to let media to flow across it. Traffic management based on application protocols has been pioneered by companies such as Cisco with their NBAR (Network-based Application Recognition) devices [1], and Palo Alto Networks with their application-based firewalls [2]. Today these traffic inspection facilities are available in every modern network security device as the binding port/protocol no longer holds.

The need to increasing visibility on network traffic, has triggered the development of DPI (Deep Traffic Inspection) libraries that replaced the first generation of port-based tools. The analysis of the packet payload is sometimes criticised as in some countries it is forbidden and also because it can use significant computing resources in order to be performed [15]. This has triggered the development of statistical-based tools based on Machine-Learning Algorithms (MLA) [13] that instead of inspecting the packet payload, rely on statistical protocol properties such as packet size distribution and intra-packet arrival time. Although some authors claim these

algorithms provide very high detection accuracy [3, 4], real-life tests [5, 6, 10, 12, 14] have demonstrated that:

- Such protocols are able to classify only traffic in a few categories (tenth compared to hundred of DPI libraries) that makes them unsuitable for users who need fine protocols detection granularity.
- On some tests they can provide a high-percentage of incorrect results, making them usable only for passive traffic analysis, and unsuitable for blocking traffic where high-reliability is compulsory.

The need to develop an efficient and open-source library for DPI has been the motivation for this work. Efficient means that such library can be effectively used to monitor 10 Gbit traffic on commodity hardware without using costly network acceleration hardware. Open-source is compulsory because:

- Commercial DPI libraries are very expensive both as one-time license fee and maintenance costs. Sometimes their price is set based on the yearly revenues of the companies that are using it rather than on a fixed per-license fee.
- Closed-source DPI libraries are often not end-user extensible. This means that users willing to add custom protocols support need to request these changes to the library manufacturer, process that introduces additional costs as well can add latency to the project until the manufacturer schedules such changes.
- Open-source tools cannot use commercial DPI libraries as they are subject to NDA (Non-Disclosure Agreement) that makes them unsuitable to be mixed with open-source software and included into the operating system kernel.

The need to create an efficient open-source DPI library for network monitoring has been the motivation for this work. As DPI has been sometimes mixed with user activity inspection and traffic interception, the source code availability is very important as it allows library users to inspect the code and thus see if it contains trojans or malware. The rest of the paper is structured as follow. Section 2 describes **xxxxxxxx** Finally, Section 6 concludes the paper.

## II. Background and Motivation

DPI is the act of inspecting the data part of a packet (packet payload) at an inspection point. Payload inspection is performed for various reasons including application protocol identification, traffic pattern identification and metadata (e.g. user name) extraction. Some libraries such as proprietary solutions from iPoque, QOSMOS, and Vineyard carry on all activities, whereas other such as libprotoident [7], UPC [8], and L7-filter [9] limit their scope to protocol identification. Protocol detection can be implemented using pattern matching or with specialised protocol decoders. The first approach is slow due to the use of regular-expressions, and error-prone as:

- It does not reconstruct packets in 6-tuple flows (VLAN, Protocol, IP/port source/destination) thus missing cross-packet matches.
- Searching for patterns on un-decoded payload can lead to search data out of context (e.g. an email including an excerpt from a HTTP connection might be confused with web-traffic) or miss matches when specific packet fields (e.g. NetBIOS host name) are encoded.

Selecting the right DPI library depends on what applications using the library expect in terms of features. In our case we focus on network traffic monitoring that can range from passive packet analysis to active inline packet policy. Key features of a DPI library must include:

- High-protocol detection reliability, so that it can be used in inline applications that can apply specific policies to different application protocols.
- Library extensibility in order to both accommodate new protocols and configure sub-protocols at runtime. New protocols appear from time to time, and also existing protocols (e.g. the Skype protocol has changed significantly since it has been acquired by Microsoft) can change over time, thus requiring permanent library maintenance.
- Availability under an open source license, for using it with existing open-source applications, and embedding into the operating system kernel. As already discussed, full source code availability is compulsory for satisfying the concerns on privacy issues.
- Extraction of some basic network metrics (e.g. network and application latency) and metadata (e.g. DNS query/response) that can be used for monitoring applications to avoid decoding the same packet twice, both in the DPI library and in the monitoring application.

For this reason, we focused our interests on DPI implemented using protocol decoders in order to have a reliable protocol recognition, as well the ability to extract selected metadata parameters that can then be used by applications using the DPI library. Libraries used only for protocol detection such as libprotoident, although released under an open-source license, are limited in scope as they do not perform any metadata extraction as it analyses only the first 4 bytes of payload in each direction to detect the protocol. As commercial DPI libraries cannot be used due to their high license cost and lack of code access, we focused on OpenDPI, an open-source predecessor of the commercial iPoque PACE (Protocol and Application Classification Engine) and no longer maintained by its developers. OpenDPI has been designed to be both an application protocol detection and metadata extraction library. Being it unmaintained, the library did not include any modern protocol (e.g. Skype) and also its internals were not cleanly designed as it was probably used to prototype features that would have been then implemented in the commercial product. The main advantage of OpenDPI was the fact that it was distributed under the GPLv3 license that allows developers to include it in software applications without being bound to an NDA or other restrictions typical of commercial DPI products. Furthermore an open-source license allows the code to be inspected, key requirement when the packet payload is instructed and potentially private information might leak. For this reason we have decided to use OpenDPI as initial core for nDPI, and then make specific library changes to address some issues we have identified.

### A. *From OpenDPI to nDPI*

The OpenDPI library is written in C language and it is divided in two main components:

- The core library that is responsible for handling raw packets, decoding layer three/four, and extracting basic information such as IP address and port.

- The plugin dissectors that are responsible for detecting the ~100 protocols supported by OpenDPI.

nDPI has inherited this two-layer architecture but it has addressed several issues present in the OpenDPI design:

- The OpenDPI library was designed to be extensible, but in practice the data structures used internally were pretty static. For instance many datatypes and bitmaps, used to keep the state for all supported protocols, were bound to specific sizes (e.g. 128 bits) making in practice the library limited in terms of protocols being detected.

- Whenever a protocol was detected, the library tried to find further protocol matches instead of just returning the first match. The result was a performance penalty without a real need of requiring extra detection work.

- No visibility of encrypted protocols, making the library blind with respect to protocols such as HTTPs. While encryption is designed to preserve privacy and thus it is normal that DPI libraries struggle with it, it is important that additional information is reported in order to give an idea of the nature of the information carried on a specific connection.

- OpenDPI was not designed to be a reentrant (i.e. thread-safe) library. This required multi-threaded applications to create several instances of the library or add semaphores in order to avoid two threads to modify the same data at the same time. Reentrancy support has been a great change in the library and in many of its component, as it has required to change many data structures in order to keep the library state per-thread. Not to mention the many global variables that were used in OpenDPI and that prevented the library to be used by multiple thread even using semaphores.

- In many parts, the OpenDPI code was not designed nicely as it contained a lot of byzantinism, and some design choices were questionable. For instance the library was performing a lot of initialisations per-flow, instead of doing them once. This means that applications using the library had to pay an unnecessary performance ticket whenever a new connection was passed to OpenDPI application detection. We believe that these design choices might have been due to the fact that the library was probably used as prototype/ playground for the commercial version of library, and so overtime the code needed some cleaning to remove , without polishing the code

- The protocol dissection was pretty “flat”. This means that whenever there is a new connection to be analysed, the library was not trying to start applying the dissectors in based on the probability of matching. For instance if there is a connection on TCP port 80, OpenDPI was not trying the http dissector first, but it was applying dissectors in the same order as they were registered in the library.

- The library was not configurable at runtime at all, and the only way to define new dissectors was to code them in C. While this is usually good for efficiency reasons, sometimes it is not a flexible approach. For instance if a given user needs to define a custom protocol Y as TCP/port-X, instead of changing the library it would have been enough to let the library accept a configuration directive at runtime. OpenDPI assumes that the library must have a dissector for all supported protocols, although this is a strong assumption in reality. In particular, on closed-environments such as a LAN or a production factory, sometimes specific hosts use proprietary/custom protocols that flow on specific ports/ protocols, and it is more convenient for the user to detect them from the packet header rather than from its payload.

- OpenDPI has not been designed to extract any metadata from analysed traffic. In one hand this is good for the privacy, but on the other hand it requires monitoring applications to decode once more the application traffic in order to extract basic information such as the URL from HTTP traffic. Reporting this information does not add any overhead to the library as it is decoded anyway when parsing the packet payload.

In essence the OpenDPI has been a good starting point for nDPI as this avoided us to start from scratch. On the other hand, many components of the original library have been changed in order to address the issues we have identified. This has been a prerequisite in order to create an efficient DPI library in particular before extending the set of protocols supported by nDPI. In fact the number of protocols recognised has an impact on both DPI detection performance and protocol recognition. The more protocols are recognised, the more time is spent on detection whenever a specific traffic pattern is not recognised and thus all the possible protocol decoders have to be tested for match. This means that DPI libraries supporting many protocols in specific situations can be slower with respect to libraries which sport much fewer protocols. Another impact on performance is due to metadata extraction: the richer is the set of attributes extracted, the slower is the processing. Although specific activities such as string and pattern matching can be accelerated on specialised hardware platforms such as Cavium and RMI, or using GPUs, we have decided not to use any of these cards, in order to let the library operate on all hardware platforms.

nDPI has been designed to be used in applications that need to detect the application protocol used in monitored traffic. It s focus is on Internet traffic, so all the available protocol dissectors support standard protocols (e.g. HTTP and SMTP) or selected proprietary protocols (e.g. Skype or Citrix) that are popular across the Internet community. Although nDPI can extract selected metadata (e.g. HTTP URL) from analysed traffic, it has not been designed as a library to be used in fields such as lawful interception or data leak prevention, as its primary goal is to characterise network traffic. Similar to OpenDPI, nDPI can be used both inside the Linux kernel and in user-space applications. As portability is one of the primary goal for open-source applications, nDPI has been ported to most operating systems including Linux, Windows, MacOS X and the BSD family. In terms of CPU architectures, it runs on x86 (32 and 64 bits), MIPS and ARM processors.

## III. nDPI Design and Implementation

In nDPI an application protocol is defined by a unique numeric protocol Id and by a symbolic protocol name (e.g. Skype). Applications using nDPI will probably use the protocol Id whereas humans the corresponding name. In nDPI a protocol includes both network protocols such as SMTP or

DNS, and communications over network protocols. For instance in nDPI Facebook and Twitter are two protocols, although from the network point of view they are communications from/to Facebook/Twitter servers used by the two popular social networks. A protocol is usually detected by a protocol decoder written in C, but it can be defined also in terms of protocol/port, IP address (e.g. traffic from/to specific networks), and protocol attributes. For instance the Dropbox traffic is identified by both the dissector for LAN-based communications and by tagging as Dropbox the HTTP traffic on which the 'Host' header field is for hosts such as '*.dropbox.com'. As explained later in this section, the nDPI library comes with detection of over 170 protocols, but it can also be extended at runtime using a configuration file for further extending it.

The nDPI library inherits the same design of OpenDPI, where the library code is used for implementing general functions, and protocol dissection is implemented in plugins. All the library code is now fully reentrant, meaning that applications based on nDPI do not need to use locks or other techniques to serialise operations. All the library initialisation is performed once at startup, without a runtime fee when a new packet needs to be dissected. nDPI expects that the library caller has already divided the packet in flows (i.e. set of packets with the same VLAN, protocol, IP/port source/destination), and that the packet has been decoded up to layer three. This means that the caller has to handle all the layer-2 encapsulations such as VLAN and MPLS, by leaving to nDPI the task of decoding the packet from the IP layer up. nDPI comes with a simple test application named pcapReader.c[1] that shows how to implement packet classification in flows and provides functions for efficient flow processing. The protocol dissectors are registered with attributes such as the default protocol and port. This means for instance that the HTTP dissector specified the default TCP/80, and the DNS dissector TCP and UDP on port 53. This practice has two advantages:

- Packet belonging to an unclassified flow (i.e. a flow for which the application protocol has not been detected yet) are passed to all dissectors registered starting from the most likely one. For instance a TCP packet on port 80, is first passed to the HTTP protocol and then if not detected is passed to the remaining registered dissectors. Of course only the dissectors for TCP protocols are considered, whereas those for non-TCP protocols are not considered for this packet. This solution in average reduces the number of dissectors that are tested, and decreases the matching time as the most likely dissector is checked first. Please note that this optimisation does not prevent from detecting HTTP on non-standard ports, but it increases the detection performance by first testing the most likely case.

- When a flow is unclassified (e.g. nDPI has tried all dissectors but none has matched), nDPI can guess the application protocol by checking whether there was a protocol registered for the protocol/port used by the flow. Note that a flow can be unclassified not just for limitations of the protocol dissectors, but also because not all flow packets where passed to nDPI. A typical example is the case when nDPI has to dissect packets belonging to a flow whose beginning has not been analysed (e.g. nDPI has been activated after the flow start).

The protocol recognition lifecycle for a new flow is the following:

- nDPI decodes the layer 3 and 4 of the packet.

- In case there is a dissector registered for the packet protocol/port, such dissector is tried first.

- In case of no match, all the registered dissectors for the packet protocol (i.e. in case of a UDP packet, all UDP dissectors are tried, but no non-UDP dissector is considered) are tried. If a dissector cannot match a packet, it has two options: either the match failed because the analysed packet will never match (e.g. a DNS packet passed to the SNMP dissector), or it failed but it might be that future packets will match. In the former case, the dissector will not be considered for future packets belonging to the same flow, whereas in the latter case the dissector will still be considered.

- Protocol detection ends as soon as a dissector matches.

A typical question of nDPI users, is how many packets we need to detect the application protocol, or to decided that a given flow is unknown. In our experience we have learnt that the answer depends on the protocol. For most UDP-based protocols such as DNS, NetFlow or SNMP one packet is enough to make this decision. Unfortunately there are other UDP protocols such as BitTorrent whose signature might require up to 8 packets in order to be detected. This leads us to the rule of thumb that in nDPI at most 8 packets per direction are enough to make a decision.

### *A. Handling Encrypted Traffic*

Like it or not, the trend of Internet traffic is towards encrypted communications. Due to security and privacy concerns, HTTPS is slowly replacing HTTP not just for secure transactions but also for sending tweets, sending messages to mobile terminals, posting notes and performing searches. Identifying this traffic as SSL is not enough, but it is necessary to characterise it better. When using encrypted communications, the only part of the data exchange that can be decoded is the initial key exchange. nDPI contains a decoder for SSL that extracts the host name of the contacted server. This information is placed in the nDPI flow metadata similar to what happens with in the HTTP decoded when extracting the server host name from the 'Host:' HTTP header. With this approach we can:

- Identify known services and tag them according to the server name. For instance an encrypted communication towards a server named 'api.twitter.com' is twitter, 'maps.google.com' is Google maps, or '*.whatsapp.net' is the WhatsApp messaging protocol.

- Discover self-signed SSL certificates. This information is important as it might indicate that the connection is not safe, not in terms of data leak, but in terms of the activity behind the communication. For instance symmetric (i.e. the traffic is not predominant in one direction such as in HTTPS, where the client sends little traffic with respect to the traffic sent by

[1] The application source code is available at https://svn.ntop.org/svn/ntop/trunk/nDPI/example/pcapReader.c

the server) long standing SSL connections with self-signed certificates often hide SSL VPNs.

As described later in this section, nDPI contains a configuration for many known protocols that are discovered using the above technique. In addition, it is possible to add at runtime a configuration file that further extends the set of detected protocols so that new ones can be defined without changing the protocol dissector. Please note that with the advent of CDN (Content Delivery Networks) this is probably the only way of identifying the application protocol as at any given time the same server (identified with a single IP address) can deliver two different services provided by two customers using the same CDN. As fallback nDPI can identify specific application protocols using the IP address. For instance nDPI detects many Apple-provided services such as iTunes and iMessage, but in addition to that it marks as Apple (generic protocol) all communications that have not been identified more in details by the available dissector but that have been exchanged with the Apple-registered IP addresses (i.e. 17.0.0.0/8).

### B. *Extending nDPI*

As previously explained, nDPI users can define new protocols not just adding a new protocol dissector but also providing a configuration file at runtime. The file format is the following.

```
#  Format:
#  <tcp|udp>:<port>,<tcp|udp>:<port>,.....@<proto>

tcp:81,tcp:8181@HTTP
udp:5061-5062@SIP
tcp:860,udp:860,tcp:3260,udp:3260@iSCSI
tcp:3000@ntop

#  Subprotocols
#  Format:
#  host:"<value>",host:"<value>",.....@<subproto>

host:"googlesyndacation.com"@Google
host:"venere.com"@Venere
host:"kataweb.it",host:"repubblica.it"@Repubblica
```

1. nDPI Configuration File.

New protocols are defined by name. In case nDPI detects that a protocol name is already defined (e.g. in the above example SIP and HTTP are handled by the native dissector), the configuration file extends the default configuration already present in nDPI. For instance in the previous example, whenever nDPI sees TCP traffic on port 81 or 8181 it tags it as HTTP. I In addition to that, nDPI can identify a protocol also by means of strings that are matched against metadata extracted from the nDPI flow such as HTTP Host and SSL certificate server name. The defined strings are stored on a automata based on the Multifast[2] library that implements string matching according to the Aho-Corasick algorithm. This library is quite efficient: at startup the automata creation takes little time (i.e. almost instantaneous with tenth of strings, or some seconds with hundred thousand strings), then this library configured with hundred thousand strings performs over 10 Gbit during search.

## IV. nDPI Validation

There are recent papers that compare the nDPI accuracy in terms of protocol detection against other DPI toolkits. Their conclusion is that “nDPI and libprotoident were successful at correctly classifying most (although admittedly not all) of the applications that we examined and only one of the evaluated applications could not be classified by both tools” [12], and “the best accuracy we obtained from nDPI (91 points), PACE (82 points), UPC MLA (79 points), and Libprotoident (78 points)” [5]. These tests have demonstrated that nDPI is pretty accurate, even more accurate than PACE that is the commercial version of the old OpenDPI library. We are aware that nDPI has some false positives with Skype and BitTorrent due to the use of heuristics. In the latest nDPI version (svn revision 7249 or newer) we have decided to remove the use of these heuristics, so that we have basically removed false positives at the cost of slightly increasing the number of flows undetected when using these two protocols.

As there are many extensive tests on nDPI protocol detection accuracy, on this paper we have decided to focus on nDPI performance. For this reason we have developed an application named pcapReader[3] that can both capture from a physical network device and read packets from a pcap file. In order to test nDPI on a physical network at 10 Gbit, we have used the test application on top of PF_RING [16], that allows applications on commodity hardware to process packets in RX/TX at 10 Gbit line rate any packet size. For our tests we have used a pcap file of over 3 million packets, captured on a heterogeneous environment thus including both LAN protocols (e.g. NFS and NetBios) and Internet protocols (e.g. Skype and DropBox). For our tests we have used a PC running Ubuntu Linux 13.10 (kernel 3.11.0-15) on a 8 core Intel i7 860. We have bound the application to a single core, in order to test it in the worst cans and see how the application can scale when using multiple cores. The test outcome is depicted below:

```
# taskset -c 1 ./pcapReader -i ~/test.pcap

Using nDPI (r7253)

pcap file contains

IP packets: 3000543 of 3295278 packets

IP bytes: 1043493248(avg pkt size 316 bytes)

Unique flows: 500

nDPI throughout: 3.42 M pps / 8.85 Gb/sec

Guessed flow protocols: 82
```

1. nDPI Validation Test Outcome.

The test outcome has demonstrated that the test application processes packets at an average speed of 3.5 Mpps / 8.85 Gbps using a single core. As the test pcap file using during the test has been captured on a real network, it contained some flows that already begun at the time the packet capture started. nDPI

[2] http://multifast.sourceforge.net

[3] https://svn.ntop.org/svn/ntop/trunk/nDPI/example/pcapReader.c

detects a flow protocol looking at the initial flow packets, thus there are some flows that are undetected due to this. For undetected flows, nDPI can guess the protocol by using the flow protocol/port registered during startup or it can leave the flows undetected. When using this test application over PF_RING DNA on a 10 Gbit Intel adapter, it is possible to use the network driver with hardware flow balancing. This way we can start one instance of the test application per virtual queue, binding each instance to a different core. As per-core the application can process more than 8 Gbps, if the traffic can be reasonably balanced by using two cores we can DPI traffic on a 10G link full of real network traffic using a low-cost system as the one used in our tests.

## V. CONCLUSIONS

XXXXX

## CODE AVAILABILITY

This work is distributed under the GNU GPLv3 license and is freely available in source format at the ntop home page https://svn.ntop.org/svn/ntop/trunk/nDPI/ for both Windows and Unix systems including Linux, MacOS X, and FreeBSD. The PF_RING framework used during the validation phase is available from https://svn.ntop.org/svn/ntop/trunk/PF_RING/.

## ACKNOWLEDGMENT

Our thanks to Italian Internet domain Registry that has greatly supported the development of nDPI, Alexander Tudor <alex@ntop.org> and Filippo Fontanelli <fontanelli@ntop.org> for their help and suggestions.

## REFERENCES

1. Cisco, Network Based Application Recognition (NBAR), 2008.
2. Palo Alto Networks, Next-Generation Firewall Overview, 2011.
3. S. Ubik, P. Zejdl, Evaluating application-layer classification using a Machine Learning technique over different high speed networks. 2010 Fifth International Conference on Systems and Networks Communications, IEEE 2010, pp. 387–391.
4. J. Li, S. Zhang, Y. Lu, Z. Zhang, Internet Traffic Classification Using Machine Learning, Proceedings of CHINACOM ’07, August 2007.
5. T. Bujlow, V. Carela-Español, P. Barlet-Ros, Comparison of Deep Packet Inspection (DPI) Tools for Traffic Classification, Technical Report, Version 3, June 2013.
6. M. Dusi, F. Gringoli, and L. Salgarelli, Quantifying the accuracy of the ground truth associated with Internet traffic traces, International Journal of Computer and Telecommunications Networking, Vol. 55 Issue 5, 2011
7. S. Alcock, R. Nelson, Libprotoident: Traffic Classification Using Lightweight Packet Inspection ,Technical report, University of Waikato. http://www. wand. net. nz/publications/lpireport, 2013.
8. L7-filter, http://l7-filter.sourceforge.net.
9. T. Bujlow, T. Riaz, J.M. Pedersen, A Method for classification of network traffic based on C5.0 Machine Learning Algorithm, in proceedings of ICNC’12, pp. 244–248, February 2012.
10. N. Cascarano, L. Ciminiera, and Fulvio Risso, Optimizing Deep Packet Inspection for High-Speed Traffic Analysis, Journal of Network and System Management, 2011.
11. P.M. Santiago del Rio, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, J. Aracil, Wire-speed statistical classification of network traffic on commodity hardware, Proceedings of IMC 2012, 2012.
12. S. Alcock, R. Nelson, Measuring the Accuracy of Open-Source Payload-Based Traffic Classifiers Using Popular Internet Applications, IEEE Workshop on Network Measurements (WNM), 2013.
13. T. Bujlow, R. Tahir, J. Myrup Pedersen, A method for classification of network traffic based on C5. 0 Machine Learning Algorithm, Proceedings of *ICNC 2012*, 2012.
14. R. Goss, R. Botha, Deep Packet Inspection—Fear of the unknown, Proceedings of *ISSA 2010*, 2010.
15. M. Avalle, F. Risso, R. Sisto, Efficient Multistriding of Large Non-deterministic Finite State Automata for Deep Packet Inspection, Proceedings of ICC 2012, 2012.
16. F. Fusco, L. Deri, High Speed Network Traffic Analysis with Commodity Multi-core System, Proceedings of IMC 2010 Conference, November 2010.