nDPI: Open-Source High-Speed Deep Packet Inspection

Luca Deri [1, 2], Maurizio Martinelli [1], Alfredo Cardigliano [2]
IIT/CNR [1]
ntop [2]
Pisa, Italy
{luca.deri, maurizio.martinelli}@iit.cnr.it, {deri, cardigliano}@ntop.org

***Abstract*—Network traffic analysis has been traditionally limited to packet header, because the transport protocol and application ports were usually sufficient 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 creating a motivation for creating tools and libraries for network protocol classification.**

This paper covers the design and implementation of nDPI, an open-source library for protocol classification using both packet header and payload. nDPI has been extensively validated in various monitoring projects ranging from Linux kernel protocol classification, to analysis of 10 Gbit traffic, reporting both high protocol detection accuracy and efficiency.

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

I. INTRODUCTION

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

Protocol identification is often not reliable even when a static port is used. A case in point is TCP/80 used for HTTP. Originally HTTP was created to carry web-related resources such as HTML pages and decorative content. However, its extensibility (in no small part due to its header flexibility and MIME type specification) along with its native integration in web browsers HTTP is now often used to carry non web-related resources. For instance, it is now the de-facto protocol for downloading/uploading files, thus replacing the FTP (File Transfer Protocol), which was designed specifically for that purpose. The pervasive use of HTTP and its native support of firewalls (i.e. a firewall recognises and validates the protocol header), has made HTTP (and its secure counterpart HTTPS) the ideal developer choice when creating a new protocol that has to traverse a firewall without restrictions. Many peer-to-peer protocols and popular applications (e.g. Skype) use HTTP as last resort when they need to pass through a firewall in case all other ports are 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 does not mean that users mostly use it for surfing the web. This protocol is extensively used by social networks, geographical maps, and video-streaming services. In other words the equation TCP/80 = web is no longer valid.

The characterisation of network protocols is required not only for creating accurate network traffic reports, but increasingly, for overall network security needs. Modern firewalls combine IP/protocol/port based security with selected protocol inspection in order to validate protocols, in particular those based on UDP (e.g. SNMP - Simple Network Management Protocol - and DNS - Domain Name System). VoIP protocols, such as SIP and H.323, are inspected for specific information (e.g. the IP and port where voice and video will flow) that allows the firewall to know what IP/ports to open to allow media flow. Cisco NBAR (Network-based Application Recognition) devices [1], and Palo Alto Networks application-based firewalls [2] have pioneered application-protocol based traffic management. Today these traffic inspection facilities are available on every modern network security device because the binding port/protocol scheme no longer holds.

The need to increase network traffic visibility has created a need for DPI (Deep Traffic Inspection) libraries to replace the first generation of port-based tools [21]. Payload analysis [22], however, uses extensive computational resource [15]. This difficulty triggered the development of statistical analysis based approaches [28] often based on Machine-Learning Algorithms (MLA) [13, 20] instead of direct payload inspection. These methods often rely on statistical protocol properties such as packet size and intra-packet arrival time distributions. Although some authors claim these algorithms provide high detection accuracy [3, 4], real-life tests [5, 6, 10, 12, 14, 19] have demonstrated that:

- Such protocols are able to classify only a few traffic categories (an order of magnitude less than DPI libraries)

and thus less suitable for fine protocol granularity detection applications.

• Some tests show a significant rate of inaccuracy suggesting that such methods may be useful in passive traffic analysis, but unlikely to be used for mission critical applications, such as traffic blocking.

These needs constitute the motivation for developing an efficient open-source DPI library where efficiency is defined by the requirement to monitor 10 Gbps traffic using solely commodity hardware (i.e. not specialised hardware needed). The use of open source is essential because:

• Commercial DPI libraries are very expensive both in terms of one-time license fee and yearly maintenance costs. Sometimes their price is set based on yearly customers revenues, rather than on a fixed per-license fee, thus further complicating the price scheme.

• Closed-source DPI toolkits are often not extensible by end-users. This means that developers willing to add new/ custom protocols support need to request these changes to the toolkits manufacturer. In essence, users are therefore at the mercy of DPI library vendors in terms of cost and schedule.

• Open-source tools cannot incorporate 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.

Although deep packet inspection has been a hot topic for a long time, beside some rare exceptions most research works have not lead to the creation of a publicly available DPI toolkit but limited their scope to prototypes or prof-of-concept tools. The need to create an efficient open-source DPI library for network monitoring has been the motivation for this work. Because DPI is a dual use technology, its users need to have source code to ensure that it is free of trojans or malware.

The rest of the paper is structured as follow. Section 2 describes the motivation of this work, and it explains how nDPI is different from its predecessor OpenDPI [17]. Section 3 covers the nDPI design and implementation. Section 4 describes the validation process, and finally section 5 concludes the paper.

II. Background and Motivation

DPI is defined as the analysis of a packet's data payload in real time (i.e. DPI processing must be faster than the traffic rate to be monitored as otherwise it would result in packet drops) at a given physical location. Inspection is motivated by various reasons including application protocol identification, traffic pattern analysis and metadata (e.g. user name) extraction. Some proprietary DPI library vendors such as iPoque, QOSMOS, and Vineyard cover all aspects, whereas others, such as libprotoident [7], UPC [8], L7-filter [9], and TIE [18] limit their scope to protocol identification [24, 25, 26].

Protocol detection may also be implemented using pattern matching or by using specialised protocol decoders. The former approach is inefficient due to the use of regular-expressions [23] and error-prone because:

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

Application drive the selection of the appropriate DPI library. We chose to focus on network traffic monitoring that can range from passive packet analysis to active inline packet policy enforcement. A DPI library must have include the following features:

- High-reliability protocol detection for inline, per application, protocol policy enforcement.
- Library extensibility is needed for new protocols and runtime in sub-protocols definition. This feature is required because new protocols appear from time to time or evolve (e.g. the Skype protocol has changed significantly since after the Microsoft acquisition). Permanent library maintenance is therefore required.
- Ability to integrate under an open source license for use by existing open-source applications and embedding into an operating system's kernel. As already discussed, full source code availability is essential to safeguard privacy.
- Extraction of basic network metrics (e.g. network and application latency) and metadata (e.g. DNS query/ response) that can be used within monitoring applications thus avoiding duplicate packet decoding, once in the DPI library and also in the monitoring application.

Our focus is therefore reliable protocol detection using protocol decoders combined with the ability to extract selected metadata parameter for the use of applications that is this library. This enables the extraction of selected metadata parameters that can then be used by applications using the DPI library. Other open-source protocol detection libraries, such as libprotoident, are limited in scope because it does not extract metadata and only analyses the first 4 bytes of payload in each direction for protocol detection. Because commercial DPI libraries could not be used a starting basis we chose OpenDPI, an open-source predecessor of the commercial iPoque PACE (Protocol and Application Classification Engine), which is no longer maintained by its developers. OpenDPI has been designed to be both an application protocol detection and metadata extraction library. Because it has been unmaintained for some time, the library did not include any modern protocol (e.g. Skype); the code was largely prototype quality and likely used as a proof of concept for the commercial product. A point in favour 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 inspected and potentially private information might leak. Our choice of OpenDPI as starting point was driven by these reasons. We then proceeded to make specific changes relating to issues we have identified. These reasons were the drivers for its choice.

We identified specific issues and made the requisite change to address them.

A. *From OpenDPI to nDPI*

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

- The core library is responsible for handling raw packets, decoding IP 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 static. For instance, many data-types and bitmaps, used to keep the state for all supported protocols, were bound to specific sizes (e.g. 128 bits) and thus limiting the number of identifiable protocols.
- 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 encrypted protocol support (e.g. HTTPs). While encryption is designed to preserve privacy and regular DPI libraries are not expected to decode the some information can be gleaned to suggest 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 multiple threads to modify the same data at the same time. Per thread library state was required to support reentrancy. This was a substantial change that touched many many data structures across most library components. OpenDPI also made wide use of global variables; this too had to change in order to make the library thread-safe and not require semaphores.
- Many parts of OpenDPI suggest problematic design choices. For instance, the library was performing much per flow initialisation instead of doing them once. The result was that applications using the library had to pay an unnecessary performance penalty whenever a new connection was passed to OpenDPI for 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.
- The protocol dissection was non-hierarchical. In other words whenever a new connection needed to be analysed, the library was not applying the dissectors based on their matching probability. 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 had no runtime configuration capability; the only way to define new dissectors was to code them in C anew. While this is usually a good policy for efficiency reasons, at times more flexibility is needed. For instance, if a given user needs to define a custom protocol Y as TCP/port-X it would be easier to have a runtime configuration directive instead of changing the library code. OpenDPI assumes that the library must have a dissector for all supported protocols, a difficult goal to achieve in reality. In particular, in closed-environments such as a LAN, specific hosts use proprietary/custom protocols that flow on specific ports/protocols; in this case 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. On one hand this preserves privacy, but on the other it requires monitoring applications to decode the application traffic again 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 sum OpenDPI has been a good starting point for nDPI because we did not have to start from scratch. Many components of the original library have been changed in order to address the issues we have identified. This was the ground work necessary to start the creation of an efficient DPI library and extending the set of supported. Not surprisingly, the number of protocols recognised has an impact on both DPI detection performance and protocol recognition. The more protocols recognised, the more time spent on detection whenever a specific traffic pattern is not identified and thus all the possible protocol decoders have to be tested for match. This means that DPI libraries supporting many protocols may be slower in specific situation than those supporting fewer. Another impact on performance is due to metadata extraction: the richer the set of extracted attributes, the slower 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 [27], we have decided not to use any of these cards, in order to let the library operate on all hardware platforms.

nDPI was designed to be used by applications that need to detect the application protocol of communication flow. Its focus is on Internet traffic, thus all the available dissectors support standard protocols (e.g. HTTP and SMTP) or selected proprietary ones (e.g. Skype or Citrix) that are popular across the Internet community. In the latter case, as protocol specifications are not publicly available, we had to create the dissectors by reverse-engineering network traffic created by their proprietary applications. Although nDPI can extract specific 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; 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 goals 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 currently 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 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 traffic dissector 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 set to '*.dropbox.com'. As explained later in this section, the nDPI library includes the detection of over 170 protocols, but it can also be further extended at runtime using a configuration file.

The nDPI library inherits some of OpenDPI design, 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 only once at startup, without a runtime penalty when a new packet needs to be dissected. nDPI expects the caller to provide the packet divided 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 and provides utility 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 specifies the default TCP/80, and the DNS dissector TCP/UDP on port 53. This practice has two advantages:

- Packets 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 dissectors for TCP protocols are considered, whereas those for non-TCP protocols are not. This solution, on average, reduces the number of dissectors that are tested, and decreases the matching time because the most likely dissector is checked first. Note that this optimisation does not prevent 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 because of protocol dissectors limitations, 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 may 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 the dissector will still be considered for future packets belonging to the same flow.
- Protocol detection ends as soon as a dissector matches.

A typical nDPI user question is the number of packets needed to detect the application protocol, or decided that a given flow is unknown. From experience we have learned that the answer is protocol dependent. For most UDP-based protocols such as DNS, NetFlow or SNMP one packet is enough to make this decision. Unfortunately there are other UDP-based 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 and messages to mobile terminals, posting notes, and performing searches. Identifying this traffic as SSL is not enough, but it is necessary to better characterise it. 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 marked as Twitter, 'maps.google.com' as Google maps, and '*.whatsapp.net' as the WhatsApp messaging protocol.

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

- Discover self-signed SSL certificates. This information is important as it might indicate that the connection is not safe, not just in terms of data leak, but also 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 the server) long standing SSL connections with self-signed certificates often hide SSL VPNs.

As described later in this section, nDPI contains internally 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 a 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 protocols not just by 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. When 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. Additionally, nDPI can also identify a protocol using 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 performs over 10 Gbps during search when configured with hundred thousand strings.

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 shown that nDPI is pretty accurate, even more accurate than PACE, the commercial version of the old OpenDPI library on which nDPI is based. We are aware that nDPI had some false positives with Skype and BitTorrent due heuristics use. In the latest nDPI versions (svn revision 7249 or newer), we have decided to remove the use of these heuristics, so that we have basically eliminated false positives at the cost of slightly increasing the number of undetected flows when using these two protocols.

As there are many extensive tests on nDPI protocol detection accuracy, this paper focuses on nDPI performance. To that end 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 Gbps, we have used the test application on top of PF_RING [16], which allows applications on commodity hardware to process packets in RX/TX at 10 Gbps line rate for 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). 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 case, 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 outcome has demonstrated that the test application processes packets at an average speed of 3.5 Mpps / 8.85 Gbps

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

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

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 detects a flow protocol by looking at the initial flow packets, so some flows are detected due to this reason. 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 Gbps Intel adapter, it is possible to use the network driver with hardware flow balancing. In this way we can start one instance of the test application per virtual queue, binding each instance to a different core. In sum 10 Gbps traffic can be inspected when balanced across two cores (the above tests show DPI at 8 Gbps using a single core) using the modestly priced commodity hardware we used in our tests.

In terms of memory usage, nDPI needs some memory to load the configuration and automata used for string-based matching. This memory used by nDPI is ~210 KB with no custom configuration loaded, that increases of ~25 KB when the configuration in Figure 1, is loaded. In addition to that, nDPI keeps per-flow information that is independent from the application protocol that will be detected and that takes ~1 KB per flow.

V. Final Remarks

This paper has presented nDPI, an open source toolkit released under GPLv3 license. It is currently able to detect more than 170 protocols including Skype, BitTorrent, and other messaging protocols. The validation test performed by third parties has demonstrated that nDPI outperforms some commercial and open-source toolkits in terms of protocol recognition accuracy. In terms of performance, using two CPU cores and commodity hardware, nDPI can handle a 10 Gbit link fully loaded with Internet traffic. This makes it suitable for scenarios where both detection accuracy and high performance are a requirement.

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. Clear Foundation, L7-filter, http://l7-filter.clearfoundation.com/.
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.
17. Y. Wei, Z. Yun-feng, L.Guo, Analysis of Message Identification for OpenDPI, Computer Engineering, 2011.
18. A. Dainotti, W. de Donato, and A. Pescapè, Tie: A Community-Oriented Traffic Classification Platform, Traffic Monitoring and Analysis, 2009.
19. S. Valenti, D. Rossi, A. Dainotti, A. Pescapè, A. Finamore, and M. Mellia, Reviewing Traffic Classification, Data Traffic Monitoring and Analysis, 2013.
20. T. T. T. Nguyen and G. Armitage. A Survey of Techniques for Internet Traffic Classification using Machine Learning, IEEE Communications Surveys & Tutorials, 2008.
21. M. Mellia, A. Pescapè, L. Salgarelli, Traffic Classification and its Applications to Modern Networks. Computer Networks, 2009.
22. F. Risso, M. Baldi, O. Morandi, A. Baldini, and P. Monclus. Lightweight, Payload-based Traffic Classification: An Experimental Evaluation, Proceedings of IEEE ICC ’08, 2008.
23. S. Kumar, P. Crowley, Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection, Proceedings of SIGCOMM ’06, 2006.
24. A. R. Khakpour, A. X. Liu. High-Speed Flow Nature Identification, Proceedings of ICDCS ’09, 2009.
25. H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, K. Lee. Internet Traffic Classification Demystified: Myths, Caveats, and the Best Practices, Proceedings of ACM CoNEXT 2008, 2008.
26. G. Aceto, A. Dainotti, W. d. Donato, A. Pescapè, Portload: Taking the Best of Two Worlds in Traffic Classification, Proceedings of INFOCOM IEEE Conference 2010, 2010.
27. N. Cascarano, P. Rolando, F. Risso, and R. Sisto, Infant: NFA Pattern Matching on GPGPU Devices, Computer Communication Review, 2010.
28. M. Crotti, M. Dusi, F. Gringoli, L. Salgarelli, Traffic Classification Through Simple Statistical Fingerprinting. ACM SIGCOMM Computer Communication Review, January 2007.