


Article

# DSM: Delayed Signature Matching in Deep Packet Inspection

Yingpei Zeng <sup>1,2,3,\*</sup> , Shanqing Guo <sup>4</sup>, Ting Wu <sup>5</sup> and Qiuhua Zheng <sup>1</sup>

<sup>1</sup> School of Cyberspace, Hangzhou Dianzi University, Hangzhou 310000, China; zheng\_qiuhua@163.com (Q.Z.)

<sup>2</sup> State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210000, China

<sup>3</sup> China Mobile (Hangzhou) Information Technology Co., Ltd., Hangzhou 210000, China

<sup>4</sup> School of Cyber Science and Technology, Shandong University, Jinan 250000, China; guoshanqing@sdu.edu.cn

<sup>5</sup> Hangzhou Innovation Institute, Beihang University, Hangzhou 310000, China; wuting@hdu.edu.cn

\* Correspondence: yzeng@hdu.edu.cn (Y.Z.)

Version December 5, 2020 submitted to Symmetry

**Abstract:** Deep Packet Inspection (DPI) is widely used in network management and network security systems. The core part of existing DPI is signature matching, and many researchers focus on improving the signature-matching algorithms. In this paper, we work from a different angle: the scheduling of signature matching. We propose a Delayed Signature Matching (DSM) method, in which we do not always immediately match received packets to the signatures since there may be not enough packets received yet. Instead, we predefine some rules, and evaluate the packets against these rules first to decide when to start signature matching and which signatures to match. The predefined rules are convenient to create and maintain, since they support custom expressions and statements and can be created in a text rule file. The correctness and performance of the DSM method are theoretically analyzed as well. Finally, we implement a prototype of the DSM method in the open-source DPI library *nDPI*, and find that it can reduce the signature-matching time about 30%~84% in different datasets, with even smaller memory consumption. Note that the abstract syntax trees (ASTs) used to implement DSM rule evaluation are usually symmetric, and the DSM method supports asymmetric (i.e., single-direction) traffic as well.

**Keywords:** DPI; Deep Packet Inspection; Delayed Signature Matching; DSM; Fast Path; Traffic Classification

---

## 1. Introduction

Deep Packet Inspection (DPI) is a type of packet processing that examines the whole packet payload but not only some packet fields like transport-layer (TCP and UDP) ports, and it is widely used in many network systems today [1–3]. For example, DPI is used in network routers for quality of service (QoS) management [4], where different protocols may have different levels of service (e.g., different bandwidths). Also, DPI is used in user profiling [5] or network auditing systems [6], to recognize which websites users are visiting and which applications users are using. This is usually for advertising [5], or government regulation (also known as lawful interception). Last but not least, DPI is also used in Firewalls [1] and Intrusion Detection Systems (IDS) [7,8] to recognize the protocols of packets and check for potential attacks.

The processing speed of DPI is quite critical, since one DPI instance usually needs to process the traffic from many devices. It is not surprising to see that different DPI products consider the processing speed (or called throughput) as an important factor. For example, commercial products like Qosmos [9]

and PACE [10] claim to handle up to 9~10 Gbps (Gigabit per second) per CPU core, and open-source product nDPI handles up to 8.85 Gbps per CPU core as well [11]. Improving the DPI performance could enable the same CPU cores to support more traffic (as an example, the mobile network traffic grew over 50 percent each year between 2014 and 2020<sup>1</sup>), or reduce the number of needed CPU cores under a known amount of traffic.

Since the core work of existing DPI is to match received packets against known signatures (or called patterns) [12], researchers proposed different methods to reduce the time spent in signature matching. First, many researchers tried to improve the exact signature-matching algorithms. Kumar et al. proposed D<sup>2</sup>FA [13] which introduces default transitions to compress the deterministic finite automaton (DFA). Bremler-Barr et al. proposed to use the repetitions in flows to improve the Aho-Corasick Algorithm [14]. Wang et al. proposed Hyperscan [15,16], which employs regex decomposition and advanced instructions on modern CPU to speed up pattern matching. Second, some researchers proposed different methods to reduce the *use* of signature matching. Doroud et al. proposed Chain [17], which tries to classify traffic based the consensus of the port-based and the machine-learning approaches first, and uses the DPI (i.e., signature matching) approach only for the left unclassified traffic. Snort [8] uses strings defined in the rules as the prerequisite for evaluating the rules, since string comparison is faster than regex evaluation [14,16].

In this paper, we focus on the scheduling of signature matching but not the signature-matching algorithms. By looking closely at the existing signature-matching process, we find that some signature-matching attempts are wasted since the needed packets have not been received yet. Thus, we propose a method named Delayed Signature Matching (DSM). In the method, a batch of rules is predefined for corresponding protocols, and all received packets are evaluated against these rules first. If the packets of a flow pass the evaluation of a rule, they could start signature matching with the signatures (i.e., using corresponding protocol parsers) defined by the rule. If the packets do not successfully pass the rules, the original processing method is used as usual. Intuitively, these rules produce *fast paths* in the signature matching (i.e., quickly finding and matching against the proper signatures). Furthermore, the rules are convenient to create and maintain, since although they have predefined format, they are text based and support custom expressions and statements. At runtime, the rules are parsed into ASTs (abstract syntax trees) to evaluate against the packets. We also analyze the correctness as well as the theoretical computation and memory reduction of the DSM method. Finally, we implement a DSM prototype in the open-source DPI library nDPI [11,18], and evaluate it with different datasets, including a dataset from a real deployment of China Mobile. It shows that with DSM the prototype has 30%~84% (more than 48% in most datasets) performance boost in the signature-matching phase. Our code is open source in Github for research usage.

This paper is an extension of our ICICS 2018 paper [6]. However, we redesign the DSM rule structure and corresponding processing algorithm to support dynamic rule evaluation. We also propose a method to reduce the memory footprint and give theoretical analysis. We carry out more experiments as well. The rest of the paper is organized as follows. In Section 2 we review related work. Then in Section 3 we use a motivating example to explain why we propose the DSM method. In Section 4 we describe the DSM method in detail. After theoretically analyzing the correctness and performance of the proposed method in Section 5, we give experiment results in Section 6, and conclude the paper in Section 7.

## 2. Related Work

We give a review on the DPI technique in general first, and then review the work closely related to the DSM method proposed in this paper.

---

<sup>1</sup> <https://www.ericsson.com/4a4e5d/assets/local/mobility-report/documents/2020/emr-q2-update-03092020.pdf>

## 2.1. DPI In General

DPI [3,12] is a popular traffic classification approach [2,19,20] that is used in many network management and network security systems, including network routers (for QoS management) [4], network auditing systems [6], Firewalls [1] and IDSs [8]. DPI is to inspect packet content to identify special strings (i.e., signatures, or called patterns) associated with an application. It is different from other traffic classification approaches like the *port-based approach* [2], which only uses transport-layer (TCP and UDP) ports to infer Internet applications, and is also different from the *statistical/machine-learning approach* [5,20–24], which usually uses packet content independent features like packet size, inter-packet time to detect the corresponding applications. While port-based approach is less accurate than DPI (since nowadays new applications may have no IANA registered ports or they may reuse the known ports of existing applications) [12], supervised machine-learning approaches have achieved comparable results to DPI [2,20]. Some commercial (e.g., Qosmos [9] and ipoque PACE [10]) and open-source (e.g., nDPI [11,18]) DPI products may use these different traffic classification approaches at the same time [25], and they may also use the word “DPI” to name the whole engine [9,10] since DPI is the major component in the system.

There are many existing DPI systems [4,7–11,26,27], and some surveys are available as well [3,12,28]. The DPI systems can be roughly classified [28] into *pattern-only* (e.g., only using signature string or regex) [16,26,29] and *hybrid* [4,7,9–11] (i.e., combining pattern and code-based protocol decoding [12]) types. L7 filter [26] is a typical pattern-only DPI system, which contains many regexes defined for different protocols. However, unfortunately, the regexes of L7 filter have not been updated since 2009. nDPI [11,18] is a hybrid DPI system, and is forked from OpenDPI, which is an open-source classifier derived [11] from an early version of PACE [10] (a commercial DPI product). nDPI mainly uses code to match different protocols; however, it also supports automaton and Hyperscan [15,16] in some steps like host-name matching. nDPI is open source in Github and under active development by the ntop company. The prototype of the proposed DSM method in this paper is based on nDPI, but the method should be applicable to other DPI systems as well.

Many researchers focus on developing new algorithms to speed up the matching of signatures [13,14,16,30,31], since it is the core work of existing DPIs [12]. For example, Dharmapurikar et al. proposed to store signatures in bloom filters to implement matching in hardware [30]. Kumar et al. proposed *Delayed Input DFA* (D<sup>2</sup>FA) [13] which introduces default transition between two states that share similar transitions, to substantially reduce the required space comparing with the original DFA. Bremner-Barr et al. proposed to use the repetitions in flows to skip repeated data by modifying the Aho-Corasick Algorithm [14]. Wang et al. at Intel designed Hyperscan [15,16], which employs regex decomposition and single instruction multiple data (SIMD) instructions on modern CPU to speed up pattern matching. On the other hand, there are some new research directions in DPI as well, e.g., the privacy of DPI (detecting patterns in encrypted traffic like HTTPS [32] but do not know the content) [33–35], DPI in new environments like smart grids [36], and DPI evasion [37]. These research efforts (on signature-matching performance, privacy etc.) essentially are orthogonal to the DSM method proposed in this paper, since they focus on the exact signature-matching algorithms, and the DSM method focuses on the *scheduling* of signature matching (i.e., when and which to match) for reducing the use of signature matching. Some researchers also focus on reducing the use of signature matching, and propose similar multiple-stage ideas. We specially review them in the next subsection.

## 2.2. DPI with Multiple Stages

The DSM method proposed in this paper will pre-process the packets with defined rules before doing signature matching, which to some extent, is similar to a few existing methods that process packets in multiple stages (passes) [28]. We review these literatures and systems below.

First, some researcher proposed similar multiple-stage ideas for DPI. Karagiannis et al. proposed BLINC [21], which analyzes flows in three different levels: social, functional, and application levels. The behaviors are analyzed with increasing details in these levels. For example, the social and functional

level behaviors are reflected in the graphlets created in the application level. However, BLINC is not related to signature matching. Keralaapura et al. proposed SLTC [38], which uses two tiers to detect p2p flows. A distributed collection tier detects p2p traffic by signature matching, and forwards the information of unclassified flows to a centralized processing tier. The centralized processing tier first identifies the p2p flows using Time Correlation Metric (TCM), and then extracts payload signatures and sends them to the distributed collection tier. We can see that SLTC does not aim to reduce the use of signature matching. Moore et al. [25] proposed an evaluation procedure that applies more and more powerful classification approaches one by one to a flow, until an approach classifies it, e.g., from the port-based approach to the DPI approach with the first KB content, and to the DPI approach with entire payload. However, it not convenient to add DPI customization into the procedure of [25]. Doroud et al. proposed Chain [17] which also introduces two stages to classify traffic. The first stage is consensus-based classification, which uses two fast approaches, port-based and machine-learning approaches, to classify a flow. Only flows unclassified in the first stage are passed to the second stage, which uses a slower approach, i.e., DPI, to classify flows. Chain only examines the port field in the first stage; however, DSM can examine any field of a packet.

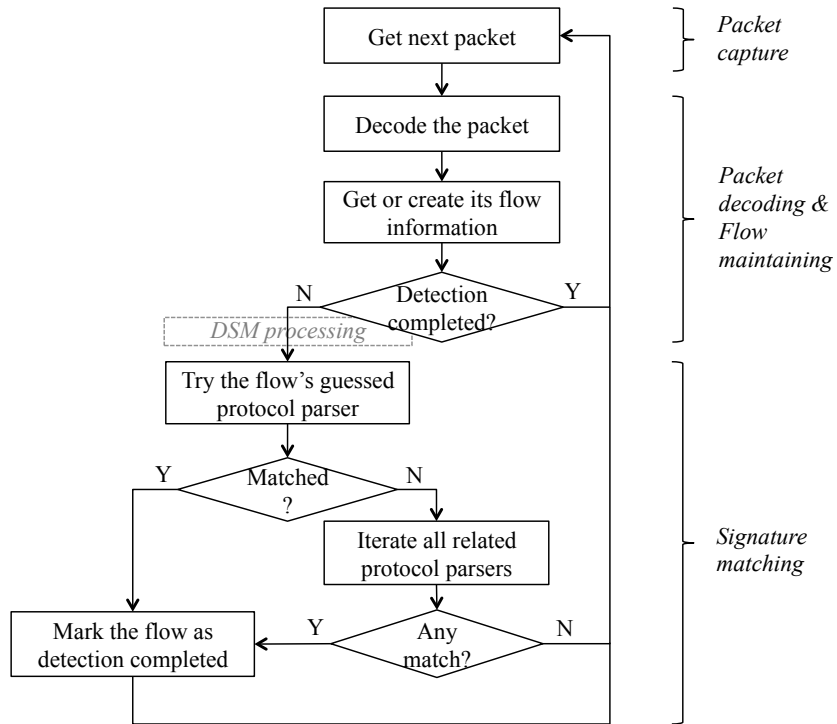
Second, some existing DPI systems (i.e., not just research prototypes) implement similar multiple-stage ideas as well. In nDPI [11,18], a `guess_protocol_id` based on the port and the protocol fields of a packet, is used to find a guessed protocol parser. The guessed protocol parser is used to match the packet first, and only if the parser fails, nDPI iterates other registered protocol parsers to match. Snort [8] automatically extracts the longest string defined in a rule, to use as the prerequisite for evaluating the rule (which means that the string should exist in the payload first). This is because string comparison is faster than regex evaluation [14,16]. Snort also has a `fast_pattern` keyword for manually changing to use some shorter but more unique string as the prerequisite for a rule. In contrast, DSM does not limit to string comparison in the first stage (e.g., may having custom statements with arithmetic). The Dynamic Protocol Detection (DPD) of Zeek (also known as Bro) [7] recommends to use port matching or signature matching to determine whether to start an application protocol analyzer for a new flow, in order to reduce useless analyzer activations. However, DPD does not support custom statements for collecting more information when determining the protocol analyzer.

### 3. Motivating Example

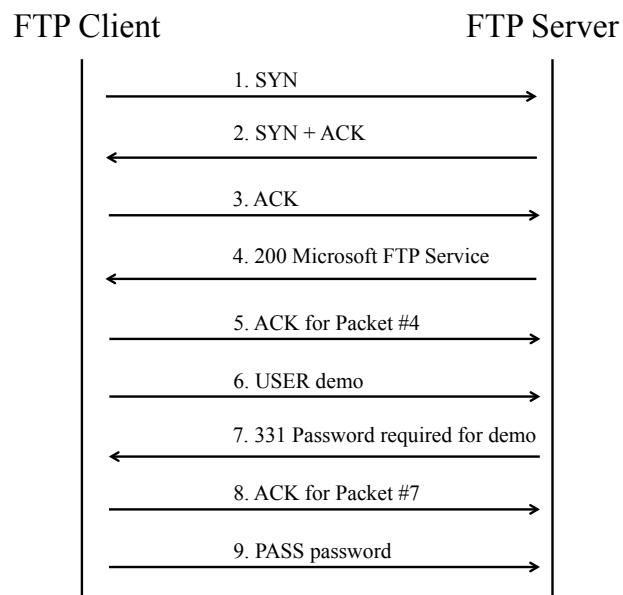
An FTP (File Transfer Protocol) example is used here to describe how the DPI engine works and why there is room to improve. We use the nDPI engine [11,18] for example, and other DPI engines like [26] are similar in general. The core workflow of the engine is drawn in Fig. 1. It could be divided into three kinds of work (i.e., three phases). In *packet capture* phase, a packet is captured from the network device, or read from a pcap file etc. In *packet decoding & flow maintaining* phase, the engine decodes the packet, and checks whether the packet belongs to a flow maintained in memory. If not, it creates a new flow for the packet. If the flow has already been marked as detection completed, the engine finishes processing the packet and get the next packet. Otherwise, it continues to the *signature matching* phase, where the engine first gets the `guess_protocol_id` based on the port and the protocol fields of the packet, and tries the corresponding guessed protocol parser. If the parser matches the application of the packet, the engine marks the flow as detection completed. Otherwise, the engine has to iterate all current related (explained later) protocol parsers and check for any match.

The first 9 packets of a typical FTP (control) connection are shown in Fig. 2. The connection contains a 3-way TCP handshake, and later USER and PASS commands to authenticate the client "demo".

How the packets of the FTP connection are processed is then shown in Fig. 3. nDPI processes packets in flows and here all packets of the FTP connection belong to the same flow. The first 3 packets are processed by 10~12 protocol parsers for signature matching, since only a few parsers are interested in and register for the *TCP and no payload* type packets, and during that some parsers find the flow does not match them at all so they exclude themselves for the flow immediately. For packet #4, the



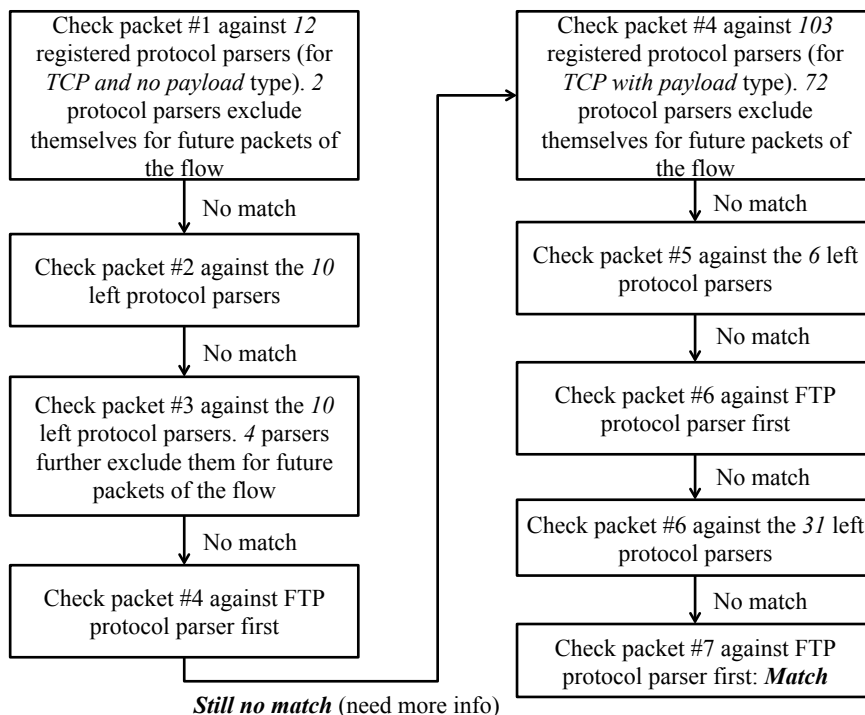
**Figure 1.** The core workflow of the nDPI engine. (The DSM processing step in the figure is for later reference in Section 4.)



**Figure 2.** The initial packets transferred during a typical FTP connection.

engine first tries the guessed FTP protocol parser based on the port 21. However, the FTP protocol parser still cannot confirm that it is an FTP flow at that time (it needs more packets to confirm). The following two packets are still no matches for detection, and only more parsers exclude themselves for the flow. Finally, the packet #7 with reply code 331 makes the FTP parser confirm it is an FTP flow and complete the detection. We can calculate that these protocol parsers are called 175 times in total to complete the detection.

There are some matching attempts wasted in the processing as we can see from Fig. 3. For example, matching the packet #4 to the 103 protocol parsers is doomed to be useless, since the critical packet #7 has not been received yet. Though nDPI uses code to match the FTP protocol, the situation is similar for regex-based DPI engines like L7 Filter [26] and Hyperscan [16]. In L7 Filter the engine keeps appending new packets to the received content buffer (2048 bytes at most) of the flow and matching the buffer against all protocols' patterns [26]. L7 Filter may use the pattern "`^220[\x09-\x0d -~]*ftp[331[\x09-\x0d -~]*password`" [39] for accurate FTP detection, then it also needs to keep matching the buffer to *all patterns* until it gets the packet #7 that contains the 331 reply code. Even for more efficient regex matching library like Hyperscan [15], which only needs to feed newly received packet into the library (i.e., in streaming mode), the whole signature database of all protocols needs to be matched to the newly received packets again and again. Thus, hyperscan also officially advises to avoid big "union" database if possible for performance<sup>2</sup>, which makes sense in the regex matching theory (an average  $O(mn/\log^{1.5}n)$  time is needed for the  $n$  text length and  $m$  pattern length case, and concatenating multiple patterns produces longer pattern) [29]. In this paper, we would like to reduce the match attempts between contents and patterns *in essence*. With the DSM method, the DPI engine only needs to match *each packet against one protocol pattern* 7 times in total (in contrast to the original 175 times) for the FTP connection example.



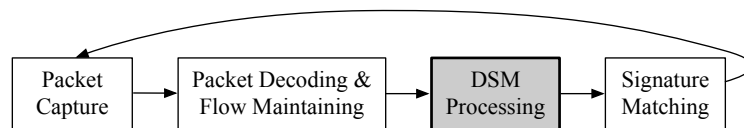
**Figure 3.** The exact processing steps of the packets in Fig. 2 in nDPI. It shows that the protocol parsers are called 175 times (12 + 10 + 10 + 1 + 103 + 6 + 1 + 31 + 1) in total for signature matching to finish the detection. We could reduce the number to only 7 with DSM.

<sup>2</sup> <http://intel.github.io/hyperscan/dev-reference/performance.html#unnecessary-databases>

## 4. Delayed Signature Matching (DSM)

### 4.1. DSM Processing Algorithm

The basic idea of Delayed Signature Matching (DSM) is that instead of immediately matching received packets to the signatures, DSM waits for *enough* packets first. In addition, DSM predefines some rules for protocols, and evaluates the received packets of a flow against the rules first, to determine whether currently received packets of the flow are enough. These rules will guide when to start signature matching and which protocol parsers to call for signature matching. The brief DPI workflow with DSM is shown in Fig. 4. We can see that the DSM processing is a phase before the original signature-matching phase, and other phases remain unchanged. As a concrete example, the position of the DSM processing in the nDPI workflow is shown in Fig. 1 in gray box.



**Figure 4.** The brief workflow of DPI with DSM.

We first briefly introduce the structure of a DSM rule before introducing the DSM processing algorithm. Rules can be stored in a text file and are read and processed into memory (discuss later in Section 4.3). The components of a DSM rule are shown in Fig. 5. The `enter_expression` represents the entering condition of a DSM rule. A flow will further use a DSM rule only after it satisfies the entering condition of the rule. The optional `per_packet_statements` are the statements that will be evaluated once when processing each packet in the flow. The `match_expression` is the expression for checking whether the packets of the flow are ready for signature matching. The `try_parser_list` is the list of protocol parsers the rule suggests for doing signature matching. We will give concrete DSM rule examples and discuss rule evaluation in detail in Section 4.2 and Section 4.3.

```

enter: enter_expression { per_packet: per_packet_statements }
match: match_expression try_parser: try_parser_list
  
```

**Figure 5.** The components of a DSM rule. The part within the braces is optional.

The DSM processing algorithm is then shown in Algorithm 1. The first step is to update the values of the built-in variables (introduce later in Section 4.2) used in the DSM rules. Then the DPI engine checks whether the `skip_dsm` flag is set to not use DSM to process the flow, or the `dsm_selected_parsers_only` flag is set to call the already selected protocol parsers only. In the two cases the engine does signature matching for the packet immediately (using the selected protocol parsers in the second case). In other cases, it needs to check the DSM rules for guidance. First, for the flows that still have not determined which rule to use, the engine iterates all rules to check whether the entering condition of a rule is met. If such a rule is found, it will be used by the flow in future. Next, for flows that have rules to use, the engine buffers the packet first, evaluates per-packet statements of the rule (to update custom variables for example), and finally evaluates the `match_expression` to check whether the flow is ready (i.e., has enough packets) to do signature matching. After the engine does signature matching for a flow as asked, the flow very likely becomes detection completed, since we create DSM rule to be so (Section 4.2). For some flows that are still not detection completed, since they may recognize the protocol type but need more packets to change detection state, we further check `is_of_protocol_type` to decide whether to continue and only call the selected protocol parsers in the future. Finally, for the cases that the protocol parsers fail to recognize the flow, or no DSM rules'

entering conditions met by the flow, the engine simply marks the flow to skip DSM in the future and does signature matching in the original way.

---

**Algorithm 1** The DSM Processing
 

---

**Input:** a packet, the packet's flow information *flow*, all DSM rules *rules*.

```

1: Maintaining built-in variables of flow for DSM rule evaluation
2: if flow.skip_dsm or flow.dsm_selected_parsers_only then
3:   /* If the flow is marked to not use DSM, or DSM has already selected parsers, just do signature matching */
4:   Do signature matching
5: else
6:   /* First, find which DSM rule to use */
7:   if !flow.determine_rule then
8:     for rule : rules do
9:       if eval(rule.enter_expression, flow) then
9:         /* the entering condition of the rule is met */
10:        flow.determine_rule = true
11:        flow.rule = rule
12:        break
13:      end if
14:    end for
15:  end if
16:  /* Second, check the rule to start signature matching */
17:  if flow.determine_rule then
18:    Buffer the packet
19:    eval(flow.rule.per_packet, flow) // evaluate per-packet update statements
20:    if eval(flow.rule.match_expression, flow) then
21:      Do signature matching (with buffered packets and flow.rule.try_parsers)
22:      /* Most flows should be detection_completed now, and for a few that are still not, further check whether parsers need more packets to complete detection */
23:      if !flow.detection_completed then
24:        if is_of_protocol_type(flow) then
25:          flow.dsm_selected_parsers_only = true
26:        else
27:          flow.skip_dsm = true
28:          Do signature matching (with buffered packets)
29:        end if
30:      end if
31:    end if
32:  else
33:    /* No rule to use */
34:    flow.skip_dsm = true
35:    Do signature matching
36:  end if
37: end if

```

---

#### 4.2. DSM Rule Creation

Since the DSM rules are in the core position of DSM processing, we next discuss how to create them with examples.

First, a few built-in variables are introduced, which could be used in the DSM rules to access the data of a packet.

- *protocol*, the protocol of the packet. For example, it could be used to check whether `protocol==tcp`.
- *server\_port*, the TCP or UDP port of the server side (the side that accepts the connection).



```

/* FTP */
enter:protocol==tcp && server_port==21
match:payload_pkt_num>=4 && payload_len>=5 &&
payload[0]=='P' try_parser:NDPI_PROTOCOL_FTP_CONTROL

/* POP3 */
enter:protocol==tcp && server_port==110
match:payload_pkt_num>=2 && payload_len>=5 &&
payload[0]=='U' try_parser:NDPI_PROTOCOL_MAIL_POP

/* SSL for HTTPS, IMAPS, SMTPS */
enter:protocol==tcp && (server_port==443||
server_port==993||server_port==465) per_packet:{if
(payload_len>=5 && (payload[0]==20||payload[0]==22||
payload[0]==23)) then {tls_pkt_num=tls_pkt_num+1;};}
match:tls_pkt_num>=1 && payload_pkt_num>=3
try_parser:NDPI_PROTOCOL_SSL

```

Figure 6. The DSM rule examples.

- `pkt_num`, stands for the number of total packets that have been received in the flow.
- `payload_pkt_num` represents the number of packets that have payload have been received in the flow.
- `payload` represents the (application layer) payload of the packet. With the operator “[ ]”, byte at any index could be accessed.
- `payload_len` represents the length of payload.

As we show later, custom variables could be used in the rules as well.

Then, the internal structure of a DSM rule is further explained here. The components of a DSM rule were already described in Section 4.1 (Fig. 5). The `enter_expression` and `match_expression` are expressions that could be evaluated to be true or false. It supports arithmetic operators (+, -, ×, /, etc.), logic AND and OR operators (&&, ||), comparison operators (<, >, ==, etc.), and other operators like ‘=’, ‘()’, and ‘[]’. The `per_packet_statements` are a list of statements and are Turing complete. Each statement could be a simple assignment statement, or flow-control statement like `if-then`, `if-then-else`, and `while-do`. The `try_parser_list` is a list of protocol parser names or protocol parser IDs.

Next, the DSM rules created for FTP, POP3 (Post Office Protocol - Version 3), and SSL/TLS (Secure Sockets Layer/Transport Layer Security) protocols are shown in Fig. 6 as examples<sup>3</sup>. For FTP, `protocol==tcp` and `server_port==21` is used as the `enter_expression`, since FTP servers usually are listening at TCP port 21. As shown in Algorithm 1, after the `enter_expression` of a rule is evaluated to be true for a flow, the `match_expression` of the rule is kept being evaluated against newly arrive packets. The rule uses `payload_pkt_num>=4 && payload_len>=5 && payload[0] == 'P'` as its `match_expression`. Satisfying `payload_pkt_num>=4 && payload_len>=5` suggests that the packet #9 (as well as #7) in Fig. 2 should have been received for the flow. The further examination of `payload[0] == 'P'` is to confirm that the first character of the “PASS” command is in the packet. If the whole `match_expression` is met, there is enough evidence that it is an FTP flow, so the rule selects the FTP protocol parser to parse all the packets received so far. As shown in Algorithm 1 if the result of the parser is matched, we mark the flow as detection completed; the flow now successfully goes through a fast path created with DSM rules. Otherwise, it is not an FTP flow and we may exclude FTP from the possible protocol list and use the original processing method. The DSM rule for the POP3 protocol is quite similar to the rule of FTP, except that we now match the “USER” command of POP3 [40] in the rule, i.e., `payload[0]=='U'`, and only 2 packets with payload would be enough for the POP3 parser to recognize the flow, i.e., `payload_pkt_num>=2`.

<sup>3</sup> The rules are created for the case that the DPI engine could process both-direction traffic, and for single-direction traffic the rules may be different.

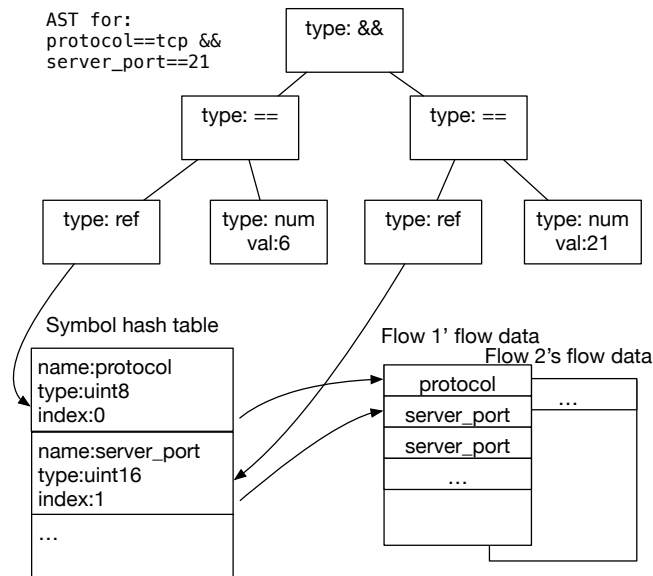
As the last example, the DSM rule created for SSL/TLS shows more flexibility in the rule creation. Similarly, the `enter_expression` is composed of `protocol` and `server_port`, though `server_port` checks against 443 for HTTPS, 993 for IMAPS (IMAP over SSL) and 465 for SMTPS (SMTP over SSL) respectively. The `per_packet_statements` part that will be evaluated for each packet introduces a custom variable named `tls_pkt_num`, which means the number of received packets that are of TLS record layer type [41] in the flow. The `tls_pkt_num` is increased when the first byte of the packet payload is 20 (`change_cipher_spec`), 21 (`alert`), 22 (`handshake`), or 23 (`application_data`). In the `match_expression` we then check whether there is at least one TLS packet received before (e.g., the `Client_Hello` message), and `payload_pkt_num` is greater than or equal to 3 (e.g., the `Client_Hello`, `Server_Hello`, and `Certificate` messages) [41]. Another thing need to be mentioned is that if later the SSL parser is selected but it does not complete the detection, we will not simply exclude SSL protocol from the flow. This is because the flow may be of SSL type, but the SSL protocol parser needs more packets to further detect its application protocol (e.g., Google). So in the `is_of_protocol_type` function we check whether it is an SSL flow by checking whether the server name has been gotten (either from the Server Name Indication (SNI) extension [42] of SSL, or from the server's certificate), or whether the SSL process has passed some stages (represented by `ssl_stage` in nDPI [18]).

### 4.3. DSM Rule Evaluation

The DSM rules are created in a text file, and are later loaded and parsed into memory by the DPI engine. This design has several advantages over creating rules in code. First, even after the initialization process, the rules could be updated during runtime by just reloading and re-parsing the modified rule file. Second, it separates the configuration from the code. For example, it enables running a compiled executable with different rule files. The network traffic usually is different in a different deployment scenario [2,21] and needs DSM rules adapted to the scenario. For example, the scenario may have only single-direction traffic for inspection, i.e., only having flows but not bidirectional flows (biflows) [2], which is common in backbones (due to the asymmetric Internet routing) [2] and network auditing (for reducing the amount of collected traffic) [6]. Third, based on our experience, creating rules in text file with predefined format is easier than creating them in code. Usually just defining two expressions is enough for a protocol in the text rule file case. Using a rule file is also a common practice in the industry. For example, Snort, [8] the popular software IDS, heavily relies on its IDS rule file.

We use ASTs (abstract syntax trees) to represent the components of a rule when parsing the DSM rules into memory, and use the data of different flows when evaluating the ASTs for different flows. Each component (the `enter_expression`, `per_packet_statements`, and `match_expression`) corresponds to an AST and a rule has at most three ASTs. We have already shown when to evaluate the ASTs in Algorithm 1 (i.e., the places calling the `eval` function), and here we just focus on the evaluation implementation. Specially, the evaluation of the ASTs for different flows needs the data of different flows. This is because for efficiency the rules and their ASTs are shared among all flows. However, the variables defined in the rules (either the built-in or custom variables) certainly have different values for different flows. Thus, as shown in Fig. 7, we allocation memory blocks for flows specially for storing the variable values, and we call the data stored in such memory blocks as *flow data*. In the symbol table (with symbol name hashed for quickly finding the corresponding entry), each entry does not store concrete variable value; instead, it stores the index (i.e., offset) of a variable in a flow's flow data. For example, the `server_port` variable's entry stores its index 1, which means that the variable value is stored at the offset 1 from the beginning of any flow's flow data. Custom variables (e.g., `tls_pkt_num` in Section 4.2) are parsed rule by rule, and each custom variable is assigned with an index that is only required to be vacant in its own rule (because the variable is accessed by its own rule only).

Flex and bison [43] are used to get the lexer and parser for DSM rules, and the ASTs are also built based on them. We choose flex and bison because they produce lexers and parsers in C programming



**Figure 7.** ASTs for a DSM rule are evaluated together with the flow data of different flows.

language (the latest ANTLR does not support the C programming language<sup>4</sup>), and the open-source nDPI library used to implement DSM prototype is written in C programming language.

#### 4.4. Discussion

*Memory footprint.* At first thought, adding the DSM processing seems to impose greater memory footprint since it needs to buffer packets until enough packets are received, but actually that is not correct in most cases, since on the other side, it also delays the allocation of the memory required for signature matching. In our prototype, we do find that the DSM processing uses less memory (see Fig. 15 in Section 6). First, the number of the buffered packets for DSM usually is not very big. For example, the required `payload_pkt_num` usually is 1~4. Also, DPI engines usually impose hard limits on the number of packets (e.g., 10 packets for a TCP flow in nDPI [18]) or on the size of the accumulated packet content (e.g., 2 KB in L7 Filter [26]) for an unclassified flow, and any flow that exceeds such limits is aborted detection directly. Thus, the extra memory used for buffering packets is not too much and is also bounded. Second, on the other hand, in DSM during the time interval between starting to buffer packets and starting to do signature matching, the memory required for signature matching is not needed yet, and can be delayed to not allocate until starting to do signature matching. For example, nDPI [18] requires a struct named `ndpi_flow_struct` (about 2.1 KB) to store the intermediate result during signature matching, L7 Filter [26] needs the 2 KB content buffer, and Hyperscan [15] requires a storage space per stream whose size is fixed at pattern compiling time. We will further analyze the memory footprint theoretically in Section 5.2.

*Rule creation considerations.* First, now the `protocol` and `server_port` are mainly used in the `enter_expression` of DSM rules, though other properties are allowed as well. This is because the port number is still a good indicator of flow's protocol. In [25], up to 69% network traffic (in bytes) could be classified with only port information [12,17]. Second, there is a trade-off between waiting more packets and detecting protocol timely when writing the `match_expression` of a DSM rule. More packets usually imply stronger evidence on the type of the flow. However, some scenarios like in Firewall or IDS [1] may require the earliest possible flow recognition. Third, protocol parsers that use the machine-learning approach to detect protocols [20] may also work with DSM rules, since features like

<sup>4</sup> <https://github.com/antlr/antlr4/blob/master/doc/targets.md>

the packet size and inter-packet time could also be evaluated in the expressions and statements of DSM rules.

*Dealing with idle DSM flows.* Note that the DSM method needs to periodically find out flows that applied DSM (i.e., having buffered packets) but are stuck for some time, and uses the original processing method to process them. This is because the flows may satisfy some DSM rule but cannot pass through it. For example, the `server_port` may be matched but the `payload_pkt_num` is not enough, and at the same time the flows do not have enough packets to eventually trigger the limit for undetected flow (e.g., 10 packets for TCP flow in nDPI [18]). If we do not use the original processing method to process them, no protocol parsers will process them even if some parser can recognize them. We set the stuck time threshold to 2 seconds, which should be sufficient for existing protocols. Also, for efficiency such periodical checking for stuck flows could be done together with the existing idle flow cleaning operation of a DPI engine.

*Relation to existing approaches and its limitation.* Existing DPI systems that have wasted signature-matching attempts due to not receiving enough packets yet, which we believe is a common problem in DPI systems like nDPI [11,18] and L7 filter [26], should benefit from using DSM. As mentioned before, existing DPI algorithms that focus on internal signature-matching strategy like [13] are orthogonal to the DSM method, and could work together with DSM to further improve the performance. A current limitation of DSM is that users need to manually create and maintain DSM rule for each protocol, even its corresponding protocol parser is already created. Fortunately, DSM rules are usually simple and text-based, thus, the extra work should be affordable.

## 5. Analysis

The analysis on the correctness and performance of the DSM method is given below.

### 5.1. The Correctness of the DSM Method

The DSM method essentially is to use the delayed packets to determine which protocol parsers to try for a flow, and there are four cases after trying the selected protocol parsers (for the case that no DSM rule is used for the flow, adding the DSM processing apparently has no impact on the correctness). We discuss them below.

The first case is that the selected protocol parsers fully detect the flow's protocol and we mark the flow as detection completed. In this case, the correctness of our method relies on the self-containing property of the protocol parsers, which means they should not need other protocol parsers to process the packets first before they can correctly recognize a flow. The self-containing property should be a reasonable assumption in reality, otherwise the codebase would be very fragile since users may change the protocol set to match for their deployment scenarios. We also confirm that all the related protocol parsers in our prototype do have the property.

The second case is that the selected protocol parsers find the flow to be not of their types, and then we could exclude the corresponding protocols and use the original processing method. The correctness of the DSM method in this case relies on the correctness of the decision that the selected protocol parsers cannot recognize the flow now and in the future. When the protocols are simple it is easy to make the decision. However if we are not sure on that, we could keep the corresponding protocols and just use the original processing method to ensure the correctness.

The third case is that the selected protocol parsers recognize the flow as their types for sure, but they need more packets to act as detection completed, and we now simply mark the flow to always use the selected protocol parsers in the future. The correctness of the DSM method in this case relies on correctly determining that the protocol parsers have recognized the flow and just need more packets for changing states. Like in TLS, we use the server name and `ssl_stage` as the indicators of the TLS detection.

The fourth case is that the selected protocol parsers are still unsure on the protocol of the flow. We intentionally want to avoid the case when creating the rules; however, there may have complicated

protocols that it is hard to bypass this situation when creating their rules. If so, we could not create DSM rules for these protocols in the first place, or if we do use DSM for them, we could simply switch to use the original processing method in the case, and the correctness is also guaranteed.

Only case #1 and case #3 are considered to be successful for the DSM method, since it successfully applies a *fast path* to the flow then and does not use the original processing method as fallback.

We note that the DSM rules could be misled by protocol masquerading mechanisms like FTE [28], since they intentionally change the signatures of protocols to mimic other protocols. Defeating them is out of the scope of the DSM method and should be the responsibility of the protocol parsers. For example, if the mimicked protocol's parser is augmented to use new detection mechanisms like entropy-based detection [44], the DSM method could still successfully detect the original protocol, given the correctness of the DSM method is assured.

## 5.2. Performance Analysis

**Computation cost.** The computation cost is analyzed first. In order to simplify the analysis, we assume a protocol has only one payload-packet type (UDP or TCP payload). We assume there are  $n$  protocol parsers interested in the payload type, and the corresponding protocol parser needs  $m$  packets to mark a flow of the protocol as detection completed. Then, for the original processing method, the number of protocol parser calls  $S_o$  can be defined as below, where  $a_i$  represents the ratio of the number of remaining parsers to the number of original parsers after processing the  $i$ th packet ( $i$  starts from 1):

$$S_o = n + a_1n + a_1a_2n + \dots + a_1a_2\dots a_{m-1}n, \text{ when } m > 1. \quad (1)$$

Now we use  $p$  to represent the possibility that the DSM method successfully matches a flow of the protocol, i.e., in either case #1 or case #3 as described in Section 5.1, and we assume the corresponding DSM rule selects  $k$  protocol parsers. Then we only need at most  $km$  protocol parser calls in the two cases (since unrelated protocol parsers may exclude themselves from the detection of the flow later). For other two cases, we at most call protocol parsers  $km + S_o$  times (when the DSM method fails and the engine uses the original method as fallback). So, with the DSM method, we could represent the upper bound of the number of protocol parser calls as below:

$$S_d = pkm + (1 - p)(km + S_o). \quad (2)$$

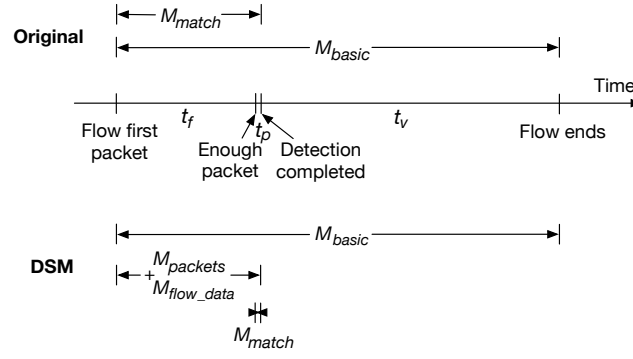
Usually we create rules that have high probability to successfully match the flows of the protocol, so  $p$  is approximate to 1 (see Fig. 14 in the Evaluation Section 6) and then  $S_d$  is approximate to  $km$ :

$$S_d \approx km, \text{ when } p \approx 1. \quad (3)$$

The value of  $km$  usually is much smaller than  $n$  ( $k$  usually is 1~2,  $m$  usually is 1~5, but  $n$  is  $> 100$  and is keep growing). Also we can see a very promising property from the equation:  $S_d$  is constant and not related to  $n$  now, which means adding new protocol parsers will not increase the computation cost, in contrast to the original processing method (Equation 1).

Note that the number of protocol parser calls does not directly reflect the real computation cost. First, the evaluation of DSM rules does add cost to the whole processing. We intentionally make the rules simple, so the added cost is small (base on our experience, it accounts for less than 1/10 of the whole signature matching time). Second, calling different protocol parsers usually has different costs, and even for the same protocol parser, calling it with different packets could have different costs. So, the analysis on the number of protocol parser calls here is just for a rough performance estimation.

**Memory footprint.** After the preliminary discussion on memory in Section 4.4, we here further analyze the memory footprint when using the DSM method. The memory usage when a flow is processed by the original method and by the DSM method is dissected separately in Fig. 8. We suppose the DPI engine could detect the protocol of the flow to simplify the analysis. We use  $t_f$  to represent



**Figure 8.** Memory footprint analysis when a flow is processed by the original method or the DSM method.

the time interval from receiving the first packet to receiving enough packets for detection, use  $t_p$  as the processing time of the DPI engine (just processing the last packet in the original method), and use  $t_v$  as the time that the flow lasts after it is detection completed. The total lifetime of the flow can be represented as  $t_{flow} = t_f + t_p + t_v$ .  $M_{basic}$  is the memory allocated for maintaining the basic flow information (e.g., the 5-tuple), and  $M_{match}$  is the memory allocated for the signature matching (e.g., the `ndpi_flow_struct` of nDPI [18]). In the DSM case, we use  $M_{packets}$  to represent the memory consumed by buffered packets, and use  $M_{flow\_data}$  to represent the memory used by the *flow data* for rule evaluation (Section 4.3).

Then the memory usage of a flow when using the original DPI method could be represented as (we use the integral of memory on time interval, but not the average memory consumption, since how long the memory is occupied is important, e.g., favoring 4KB consumption for 10 seconds rather than 2KB consumption for 1 minute):

$$M_o = \int_0^{t_{flow}} M_{basic} dt + \int_0^{t_f+t_p} M_{match} dt. \quad (4)$$

Similarly, the memory usage of the flow when using the DSM method could be represented as:

$$M_d = \int_0^{t_{flow}} M_{basic} dt + \int_0^{t_f+t_p} (M_{packets} + M_{flow\_data}) dt + \int_0^{t_p} M_{match} dt. \quad (5)$$

The difference between them is:

$$M_o - M_d = \int_0^{t_f+t_p} (M_{match} - M_{packets} - M_{flow\_data}) dt - \int_0^{t_p} M_{match} dt. \quad (6)$$

Usually  $t_p$  is very small (less than 0.1ms) comparing with  $t_f$  (from several hundred milliseconds to several seconds), so we can consider  $t_p \approx 0$ . Also,  $M_{flow\_data}$  is a small value (e.g., only 16 bytes in our prototype), and  $M_{match}$  is a constant. So we have:

$$M_o - M_d \approx \int_0^{t_f} (M_{match} - M_{packets}) dt \quad (7)$$

$$= M_{match} t_f - \int_0^{t_f} M_{packets} dt. \quad (8)$$

Suppose  $m$  packets are enough for the flow's protocol detection, and for simplicity we assume that each packet arrives at the same interval  $\frac{t_f}{m-1}$ , and has the same size  $P$ , then

$$\begin{aligned} \int_0^{t_f} M_{\text{packets}} dt &= \sum_{i=1}^m \frac{t_f}{m-1} (m-i)P \\ &= \frac{mPt_f}{2}. \end{aligned} \quad (9)$$

Then apply it to Equation (8), we get:

$$M_o - M_d = (M_{\text{match}} - \frac{mP}{2})t_f. \quad (10)$$

With the Equation 10 we could roughly estimate whether the DSM method consumes less memory than the original method (i.e., without DSM). For example, for nDPI [18] the  $M_{\text{match}}$  is `ndpi_flow_struct` about 2.1 KB, then if  $m$  is 4 and assuming the average packet size  $P$  is 420 bytes<sup>5</sup>, we can calculate  $M_o - M_d = (2100 - \frac{4 \times 420}{2})t_f = 1260t_f$ , which means that using DSM should have smaller memory footprint than without using DSM.

**Potential delay.** The potential extra delay when using DSM is analyzed here. Note that if a rule is created to strictly fit to the requirement of corresponding protocol parser (i.e., not waiting for more packets), and a flow's packets match the rule as expected, then there is no extra delay for the flow, comparing with the original processing method. This is because even using the original processing method, the same set of packets needs to be received before the flow becomes detection completed. However, if the rule is created to need more packets than the parser actually needs, DSM will have extra delay because of waiting for extra packets. Depending on the number of extra packets, DSM may have several RTT (Round Trip Time) delay. One RTT usually is at most several hundred milliseconds in the Internet, so such delay is in the order of second at most. Another case is for the flows that enter DSM process but get stuck there (not enough packets), where the extra delay is related to the stuck threshold time DSM sets (at most 2 times of the threshold). Thus, for the 2-second threshold DSM sets, the corresponding delay is 4 seconds at most.

## 6. Evaluation

### 6.1. Implementation of DSM

We implement our DSM method prototype in the popular open-source DPI library nDPI [11,18]. Our prototype is open source for research usage<sup>6</sup>. The prototype is based on a branch created from the nDPI *dev* branch with the last commit #e5a95cf (May 14, 2019). We try to keep the change to the existing nDPI code as minimal as possible. When the DPI engine starts with DSM enabled (e.g., running the built-in `ndpiReader` program of nDPI with a new `-y` option), we use flex (version 2.5) and bison (version 2.3) to parse DSM rules and build ASTs for them (described in Section 4.3) with several related files like `dsm.lex.c`, `dsm.tab.c`, and `dsm_funcs.c`. We implement the DSM processing algorithm (described in Section 4.1) mainly in the `packet_processing` method of the `ndpi_util.c` file. In the `packet_processing` method we evaluated the packets against the ASTs of the DSM rules (described in Section 4.3), to decide when to do the signature matching (i.e., calling the `ndpi_detection_process_packet` method). We do not change the code of the signature matching phase (i.e., the workflow after the `ndpi_detection_process_packet` method) except that, when calling

<sup>5</sup> [http://www.caida.org/research/traffic-analysis/AIX/plen\\_hist/](http://www.caida.org/research/traffic-analysis/AIX/plen_hist/)

<sup>6</sup> <https://github.com/zyingp/nDPI/tree/journal-dev>

**Table 1.** The properties of the datasets.

Name	Size (MB)	Time (hour)	Num. of flows	Num. of pkts	Traffic desc.
ATestbed1	79.6	23.9	30735	422966	upstream
ATestbed2	304.6	103.4	110174	1599281	upstream
BTestbed	326.4	93.2	24970	1383572	upstream
Web100	135.2	0.6	3701	222924	both
Web500	685.3	3.1	37473	1035565	both
Deployx	518.6	11.9	143314	3000000	upstream

protocol parsers (e.g., in the `check_ndpi_tcp_flow_func` method) we first check whether there are any protocol parsers selected by DSM rules for enforcing the DSM processing.

Before carrying out experiments below, several implementation errors are fixed to ensure that when DSM is enabled in nDPI (we call it the DSM+nDPI mode), the detection result of the DPI engine is exactly the same as the result of the original processing method (we call it the nDPI mode). It also proves that our theoretical correctness analysis in Section 5.1 is achievable.

## 6.2. Experiment Results

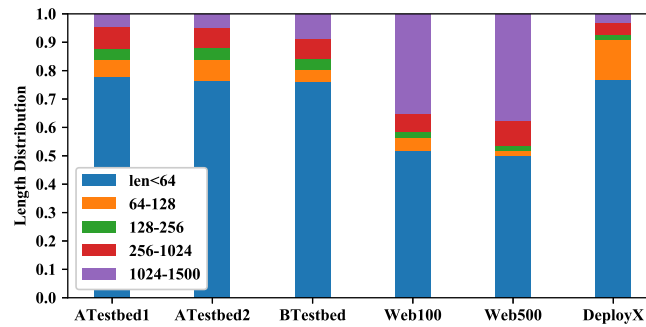
All the experiments are done on a PC installing Ubuntu 16.04 LTS, with Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz×4 and 8 GB memory. We create seven DSM rules for protocols including, SSL, HTTP, Teamviewer, FTP, POP3, RDP (Remote Desktop Protocol), and SSH (Secure Shell) (rules are slightly different when testing different datasets, for example, some rules are omitted for some datasets, and the `payload_pkt_num` threshold in a rule may be different for single-direction traffic and both-direction traffic).

Six different datasets are used for the experiments, as shown in Table 1. The datasets come from two testbeds, a web crawler, and a real-world deployment of our network auditing system [6]. We have two testbeds (named testbed A and B) for our network auditing tests, and in both testbeds *we mainly capture one side of the traffic* (i.e., upstream only, except for several special TCP ports) for auditing. The testbed A contains 2 China Mobile enterprise WiFi gateways (i.e., hotspots), with 9 computers and phones connected. The testbed B contains 1 WiFi gateway with 3 computers and phones connected (mainly computer traffic). The ATestbed1 and ATestbed2 datasets are from the same testbed A but captured at different time and have different sizes. BTestbed dataset is from the testbed B. In order to test DSM with other traffic pattern, we use a web crawler running on an iPad to browse the Alexa Top 100 and Top 500 websites<sup>7</sup> and get two full traffic traces (i.e., both upstream and downstream traffic) named Web100 and Web500 respectively. In addition, we got 3 million packets for testing when our network auditing system was deployed in a real hotel in year 2018 (named DeployX). We show the packet length distribution of the 6 datasets in Fig. 9. We can see that for upstream only traffic datasets, small packets (packet length < 256) are in the majority (over 80%), and less than 10% packets have length  $\geq 1024$ . In contrast, for both-direction-traffic datasets Web100 and Web500, over 30% packets are big packets (length  $\geq 1024$ ).

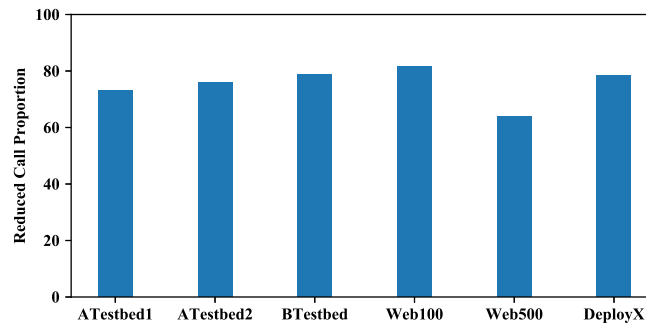
First the number that the protocol parsers are called in the two modes is checked, since the DSM method directly reduces the number to improve DPI performance. We show the result in Fig. 10. When DSM is enabled (i.e., the DSM+nDPI mode), it reduces about 64% (for Web500 dataset) to 82% (for Web100 dataset) of the calls needed in the original method (the nDPI mode). The result is reasonable considering the great reduction of protocol parser calls when a DSM rule is successfully applied to a flow.

<sup>7</sup> <https://www.alex.com/topsites>





**Figure 9.** The packet length distribution of the datasets.

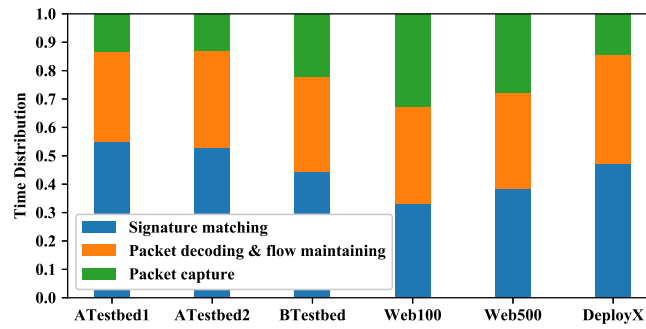


**Figure 10.** The percentage of reduced protocol parser calls, i.e.,  $(1 - \frac{\text{\#of protocol parser calls in DSM+nDPI mode}}{\text{\#of protocol parser calls in nDPI mode}}) \times 100$ , when DSM is enabled.

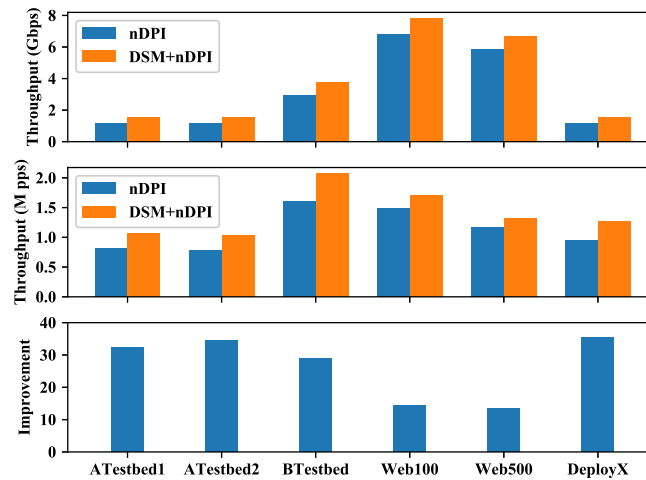
Next, it is interesting to test how the DSM method works in a real DPI program and we choose the built-in `ndpiReader` program of `nDPI`. The `ndpiReader` program supports to capture packets with `libpcap` and `DPDK` (Data Plane Development Kit), and we use `libpcap` here in order to read packets from the 6 dataset files we prepared. The `ndpiReader` program now uses an ordinary binary tree but not the better self-balancing trees like the red-black tree to maintain flows. We run the `ndpiReader` program 15 times for each dataset, discard the first 5 runs whose result may not be stable yet (since the program has large file I/O), and calculate the average result of the left 10 runs (the result is quite stable then).

Before testing the DPI throughput, we run the `ndpiReader` program to get the time consumption of each DPI phase and show the result in Fig. 11 (the explanation of the DPI phases is in Fig. 1). We can see that the signature-matching phase occupies a large proportion (ranging from 33% to 55%) of the total time consumption. The proportion also bounds how much DSM can improve the performance of the `ndpiReader` program, since DSM only improves the signature-matching phase. For example, if the signature-matching phase occupies 33% of the total time, then even if DSM reduces its time to 0, the improvement in throughput is  $(\frac{1}{1-0.33} - 1) \times 100 \approx 49\%$ . Both of the other two DPI phases have noticeable overhead in the program. For example, the packet decoding & flow maintaining phase consumes 32%~38% of the total time. For full traffic datasets, i.e., `Web100` and `Web500`, the packet capture phase consumes up to 28%~33% of the total time. We believe that with more efficient flow maintaining algorithm (e.g., using self-balancing trees) and packet capture mechanism (e.g., using `DPDK`), these two phases will occupy smaller proportion in the total time.

Then the processing throughput of the `ndpiReader` program is tested, and the result is shown in Fig. 12. The processing throughput is in the two upper subfigures, in Gbps (Gigabit per second) and Mpps (million packet per second) respectively. The throughput improvement of the DSM+nDPI mode over the nDPI mode is in the lower subfigure. We can see that the DSM+nDPI mode improves the throughput about 14%~15% for the full traffic datasets `Web100` and `Web500`, and 29%~35% for other upstream only datasets. The difference is because the signature-matching phase that the DSM



**Figure 11.** The time consumption distribution of different kinds of work in ndpiReader.

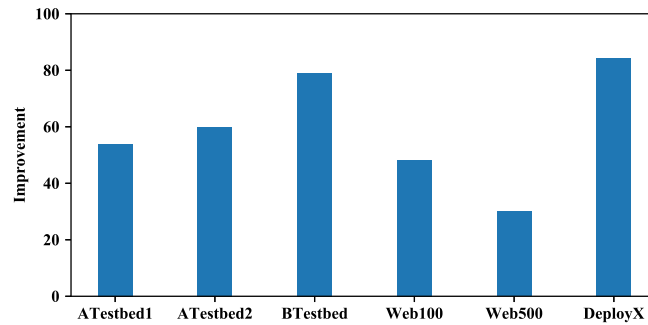


**Figure 12.** Processing throughputs of the two modes in Gbps and M pps respectively, and the improvement of the DSM+nDPI mode over the nDPI mode.

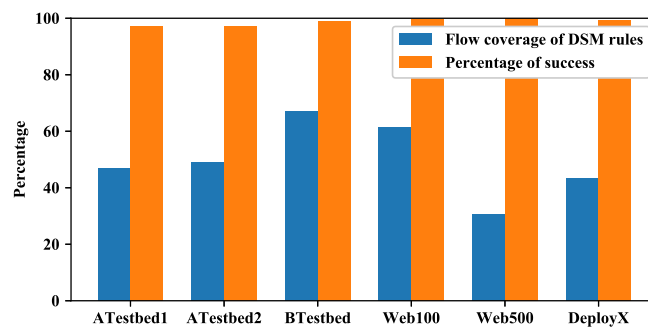
improves, has a smaller proportion of time for full traffic datasets, as we shown in Fig. 11. We could also see that the full traffic datasets have much higher throughput in Gbps but comparable throughput in M pps than other datasets, since the full traffic datasets have longer average packet length.

We know that other DPI phases amortize the DSM's improvement in the signature-matching phase, and it is interesting to check exactly the improvement the DSM brings to the signature-matching phase. The result is shown in Fig. 13. The improvement of the DSM+nDPI mode over the nDPI mode in the signature-matching phase (i.e.,  $(\frac{\text{Signature-matching time in nDPI mode}}{\text{Signature-matching time in DSM+nDPI mode}} - 1) \times 100$ ) ranges from 30% (for Web500) to 84% (for DeployX), and is greater than 48% for most datasets. We find that the performance in Web500 is slightly lower than the average because the dataset contains about 40% SSDP (Simple Service Discovery Protocol) flows, which is accidentally captured. Also, the improvement in the signature-matching phase is much less than the improvement in the protocol parser call shown in Fig. 10. This is because as we explained in Section 5.2 that the DSM method has extra cost in the DSM rule evaluation in the signature-matching phase (accounting for about 1/10). In addition, we only count the number of protocol parser calls in Fig. 10, but different protocol parsers may have different computation costs and the same protocol parser may have different costs when processing different packets (may have more than 10x difference based on our experience).

In the next experiment the effectiveness of current DSM rules is checked. The flow coverage of current DSM rules is calculated by  $\frac{\text{\#of flows that use DSM}}{\text{\#of all the flows}} \times 100$ , where a flow using DSM means that the engine tried the DSM selected protocol parsers for it. The percentage of success of the DSM rules is calculated by  $\frac{\text{\#of flows that use DSM and succeed in detection}}{\text{\#of flows that use DSM}} \times 100$ , where succeeding in detection also corresponds to the case #1 and case #3 described in Section 5.1. The result is shown in Fig. 14. We can see that the coverage of our DSM rules is 30%~67% for these datasets, which is promising since



**Figure 13.** The improvement in the signature-matching phase only (the DSM+nDPI mode vs. the original nDPI mode, including the rule-evaluation time of the DSM). The improvement ranges from 30% (for Web500) to 84% (for DeployX).

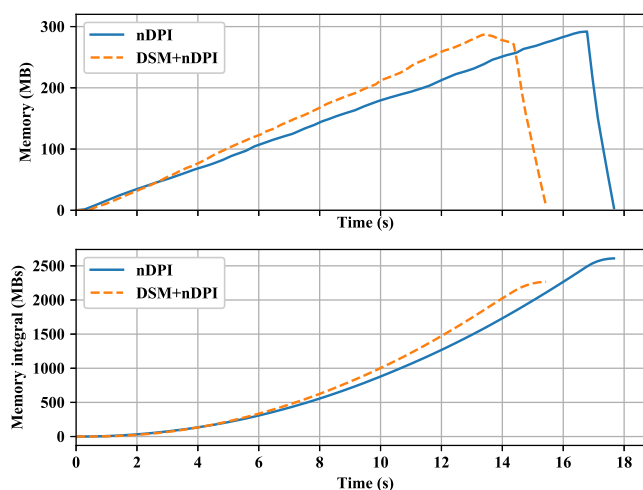


**Figure 14.** The flow coverage and percentage of success of the DSM rules in our prototype.

we only create seven rules. Most of the coverage is attributed to the DSM rules of the two protocols: SSL and HTTP. In addition, the percentage of success flows is very high, about 97.2%~99.8%. We also look into the fail cases and find there are mainly two cases. One case is that the initial packets of a flow is not captured in the dataset, for example, the Client Hello packet of a long-lived TLS flow is not captured so the flow cannot be detected. Another case is that the HTTP request line (which contains the GET or POST method) is very long and cannot fit in one packet, and the HTTP protocol parser cannot either complete the detection or recognize the protocol type in this condition. (As explained before, flows failed in the DSM method are processed by the original method as fallback.)

Finally, the memory consumption of the DSM prototype is checked with the `ndpiReader` program. We choose the `ATestbed2` dataset, where the DSM method has medium improvement (60%) in the signature-matching phase. We use the massif tool of Valgrind<sup>8</sup> to get more accurate memory consumption of the program. We only count the heap memory usage since the stack memory usage is relatively very small (only several KB). We show our experiment result in Fig. 15. The upper subfigure shows the memory size over time in both modes. We can see that with DSM the processing ends early and the peak memory is slightly smaller as well (286MB vs. 292MB). We show the memory usage integral to better illustrate the difference of the two modes. We cumulatively integrate memory usage on  $[0, t]$  time interval and use it as the value on  $t$  (specifically, we use the `integrate.cumtrapz` function in SciPy), and show the result in the lower subfigure. We can clearly see that the difference of the memory consumption in two mode become apparent over time, and is 2266MBs vs. 2609 MBs at last. Note that the memory usage is growing up with time in both cases, because the `ndpiReader` program will not free the memory of basic flow information (i.e.,  $M_{basic}$  in Fig. 8) until the program is about

<sup>8</sup> <http://valgrind.org/docs/manual/ms-manual.html>



**Figure 15.** The memory consumption comparison when processing the ATestbed2 dataset.

to exit when it is processing packets from dataset files (i.e., in offline mode, it will free in time in the online mode though).

## 7. Conclusion

In this paper we proposed a method named DSM for reducing useless signature-matching attempts. Our method is actually based on two simple ideas: delaying signature matching until receiving enough packets, and finding the probable protocol parsers (signatures) to match first. We designed the DSM processing algorithm together with what we call DSM rules, to guide when to start signature matching for a flow and which protocol parsers to use. Since it is not flexible to hardcode the DSM rules in code, we designed a rule format that enables us to create rules in text file. At runtime the DSM rules are parsed into ASTs (abstract syntax tree) and evaluated dynamically using per flow data. We also theoretically analyzed the DSM method to show when its correctness is assured, as well as its performance efficiency. We implemented a DSM prototype with nDPI and evaluated it with different datasets. The result showed that the DSM method accelerated the signature-matching phase about 84% for a real deployment dataset, and over 48% for most datasets. The DSM method is not limited to pattern based DPI, but may also be applied to machine-learning approaches as well, since the features of machine learning, like packet size and inter-packet time, could be evaluated in DSM rules.

**Author Contributions:** Conceptualization, Y.Z. and S.G.; methodology, Y.Z.; software, Y.Z.; validation, Y.Z. and S.G.; formal analysis, Y.Z. and S.G.; investigation, Y.Z.; resources, Y.Z., T.W., and Q.Z.; data curation, Y.Z.; writing—original draft preparation, Y.Z.; writing—review and editing, Y.Z. and S.G.; visualization, Y.Z.; supervision, Y.Z. and T.W.; project administration, Y.Z. and T.W.; funding acquisition, Y.Z., S.G., T.W., and Q.Z.. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported in part by the National Natural Science Foundation of China under Grant No. 61902098, and Grant No. 91546203, in part by the Key Research Project of Zhejiang Province under Grant No. 2020C01078, Grant No. 2019C01012, and Grant No. 2017C01062, in part by Major Scientific and Technological Innovation Projects of Shandong Province, China under Grants No. 2017CXGC0704, No. 2018CXGC0708, and No. 2019JZZY010132 and Qilu Young Scholar Program of Shandong University. The authors would like to thank Yanzhao Shen for his comments on the manuscript.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

- Porter, T. The Perils of Deep Packet Inspection. <https://www.symantec.com/connect/articles/perils-deep-packet-inspection>, accessed on 23 September 2020.
- Dainotti, A.; Pescapé, A.; Claffy, K.C. Issues and future directions in traffic classification. *IEEE Netw.* **2012**, *26*, 35–40.

3. Bujlow, T.; Carela-Español, V.; Barlet-Ros, P. Independent comparison of popular DPI tools for traffic classification. *Comput. Netw.* **2015**, *76*, 75–89.
4. Cisco. Network Based Application Recognition (NBAR). <https://www.cisco.com/c/en/us/products/ios-nx-os-software/network-based-application-recognition-nbar/index.html>, accessed on 23 September 2020.
5. Aceto, G.; Ciunzo, D.; Montieri, A.; Pescapé, A. Mobile Encrypted Traffic Classification Using Deep Learning: Experimental Evaluation, Lessons Learned, and Challenges. *IEEE Trans. Netw. and Service Manag.* **2019**, *16*, 445–458.
6. Zeng, Y.; Guo, S. Deep Packet Inspection with Delayed Signature Matching in Network Auditing. Proc. Int. Conf. Inf. and Commun. Security (ICICS), 2018, pp. 75–91.
7. Paxson, V. Bro: a system for detecting network intruders in real-time. Proc. 7th Conf. USENIX Security Symp., 1998.
8. Cisco. Snort - Network Intrusion Detection & Prevention System. <https://www.snort.org/>, accessed on 23 September 2020.
9. Qosmos. Qosmos DPI Engine. <https://www.qosmos.com/products/deep-packet-inspection-engine/>, accessed on 23 September 2020.
10. ipoque GmbH. DPI Engine - R&S PACE 2. <https://ipoque.com/products/dpi-engine-rsrpace-2>, accessed on 23 September 2020.
11. Deri, L.; Martinelli, M.; Bujlow, T.; Cardigliano, A. nDPI: Open-source high-speed deep packet inspection. Proc. 10th Int. Wireless Commun. and Mobile Comput. Conf. (IWCMC), 2014, pp. 617–622.
12. Finsterbusch, M.; Richter, C.; Rocha, E.; Muller, J.; Hanssger, K. A Survey of Payload-Based Traffic Classification Approaches. *IEEE Commun. Surveys Tuts.* **2014**, *16*, 1135–1156.
13. Kumar, S.; Dharmapurikar, S.; Yu, F.; Crowley, P.; Turner, J. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. Proc. ACM SIGCOMM, 2006, pp. 339–350.
14. Bremler-Barr, A.; David, S.T.; Harchol, Y.; Hay, D. Leveraging traffic repetitions for high-speed deep packet inspection. Proc. IEEE INFOCOM, 2015, pp. 2578–2586.
15. Intel. Hyperscan. <https://www.hyperscan.io/>, accessed on 23 September 2020.
16. Wang, X.; Hong, Y.; Chang, H.; Park, K.; Langdale, G.; Hu, J.; Zhu, H. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. Proc. 16th USENIX Symp. Netw. Syst. Design Implement. (NSDI), 2019, pp. 631–648.
17. Doroud, H.; Aceto, G.; de Donato, W.; Jarchlo, E.A.; Lopez, A.M.; Guerrero, C.D.; Pescapé, A. Speeding-Up DPI Traffic Classification with Chaining. Proc. IEEE Global Commun. Conf. (GLOBECOM), 2018, pp. 1–6.
18. ntop. nDPI - Open Source Deep Packet Inspection Software Toolkit. <https://github.com/ntop/nDPI>, accessed on 23 September 2020.
19. Callado, A.; Kamienski, C.; Szabo, G.; Gero, B.P.; Kelner, J.; Fernandes, S.; Sadok, D. A Survey on Internet Traffic Identification. *IEEE Commun. Surveys Tuts.* **2009**, *11*, 37–52.
20. Nguyen, T.T.T.; Armitage, G. A survey of techniques for internet traffic classification using machine learning. *IEEE Commun. Surveys Tuts.* **2008**, *10*, 56–76.
21. Karagiannis, T.; Papagiannaki, K.; Faloutsos, M. BLINC: Multilevel Traffic Classification in the Dark. Proc. ACM SIGCOMM, 2005, pp. 229–240.
22. Zhang, J.; Chen, X.; Xiang, Y.; Zhou, W.; Wu, J. Robust Network Traffic Classification. *IEEE/ACM Trans. Netw.* **2015**, *23*, 1257–1270.
23. Taylor, V.F.; Spolaor, R.; Conti, M.; Martinovic, I. Robust Smartphone App Identification via Encrypted Network Traffic Analysis. *IEEE Trans. Inf. Forensics Security* **2018**, *13*, 63–78.
24. Cao, J.; Wang, D.; Qu, Z.; Sun, H.; Li, B.; Chen, C.L. An improved network traffic classification model based on a support vector machine. *Symmetry* **2020**, *12*, 1–21. doi:10.3390/sym12020301.
25. Moore, A.W.; Papagiannaki, K. Toward the Accurate Identification of Network Applications. Proc. Passive and Active Netw. Measure. (PAM), 2005, pp. 41–54.
26. Sommer, E.; Strait, M. L7-filter. <http://l7-filter.sourceforge.net>, accessed on 23 September 2020.
27. Sommer, R.; Amann, J.; Hall, S. Spicy: A unified deep packet inspection framework for safely dissecting all your data. Proc. Annual Computer Security Applications Conference (ACSAC), 2016, pp. 558–569. doi:10.1145/2991079.2991100.

28. Dyer, K.P.; Coull, S.E.; Ristenpart, T.; Shrimpton, T. Protocol misidentification made easy with format-transforming encryption. *Proc. ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, 2013, pp. 61–72.
29. Backurs, A.; Indyk, P. Which Regular Expression Patterns Are Hard to Match? *Proc. IEEE 57th Annu. Symp. Foundations Comput. Science (FOCS)*, 2016, pp. 457–466.
30. Dharmapurikar, S.; Krishnamurthy, P.; Sproull, T.S.; Lockwood, J.W. Deep packet inspection using parallel bloom filters. *IEEE Micro* **2004**, *24*, 52–61.
31. Antonello, R.; Fernandes, S.F.L.; Sadok, D.F.H.; Kelner, J.; Szabó, G. Design and optimizations for efficient regular expression matching in DPI systems. *Comput. Commun.* **2015**, *61*, 103–120.
32. Durumeric, Z.; Ma, Z.; Springall, D.; Barnes, R.; Sullivan, N.; Bursztein, E.; Bailey, M.; Halderman, J.A.; Paxson, V. The Security Impact of HTTPS Interception. *Proc. 24th Annu. Netw. Distrib. Syst. Security Symp. (NDSS)*, 2017.
33. Sherry, J.; Lan, C.; Popa, R.A.; Ratnasamy, S. BlindBox: Deep Packet Inspection over Encrypted Traffic. *Proc. ACM SIGCOMM*, 2015, pp. 213–226.
34. Yuan, X.; Wang, X.; Lin, J.; Wang, C. Privacy-preserving deep packet inspection in outsourced middleboxes. *Proc. IEEE INFOCOM*, 2016, pp. 1–9.
35. Poddar, R.; Lan, C.; Popa, R.A.; Ratnasamy, S. SafeBricks: Shielding Network Functions in the Cloud. *Proc. 15th USENIX Symp. Netw. Syst. Design Implement (NSDI)*, 2018, pp. 201–216.
36. De La Torre Parra, G.; Rad, P.; Choo, K.K.R. Implementation of deep packet inspection in smart grids and industrial Internet of Things: Challenges and opportunities. *Journal of Network and Computer Applications* **2019**, *135*, 32–46. doi:10.1016/j.jnca.2019.02.022.
37. Wang, Z.; Zhu, S.; Cao, Y.; Qian, Z.; Song, C.; Krishnamurthy, S.V.; Chan, K.S.; Braun, T.D. SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. *Proc. 27th Annu. Netw. Distrib. Syst. Security Symp. (NDSS)*, 2020, number February. doi:10.14722/ndss.2020.24083.
38. Keralapura, R.; Nucci, A.; Chuah, C. Self-Learning Peer-to-Peer Traffic Classifier. *Proc. 18th Int. Conf. on Comput. Commun. and Netw. (ICCCN)*, 2009, pp. 1–8.
39. Bober, A.; Konieczny, J. Introduction to Layer 7-filter. [https://mum.mikrotik.com//presentations/PL10/l7\\_interprojekt.pdf](https://mum.mikrotik.com//presentations/PL10/l7_interprojekt.pdf), accessed on 23 September 2020.
40. Myers, J.G.; Rose, M.T. Post Office Protocol - Version 3. RFC 1939, 1996.
41. Dierks, T.; Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008.
42. 3rd, D.E. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, 2011.
43. Levine, J.R. *Flex and bison – Unix text processing tools*; O'Reilly, 2009.
44. Wang, L.; Dyer, K.P.; Akella, A.; Ristenpart, T.; Shrimpton, T. Seeing through Network-Protocol Obfuscation. *Proc. ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, 2015, pp. 57–69.

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Submitted to *Symmetry* for possible open access publication under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).