Deep Packet Inspection with Delayed Signature Matching in Network Auditing

Yingpei Zeng[1,2] and Shanqing Guo[3]

[1] School of Cyberspace, Hangzhou Dianzi University, HangZhou, China
[2] China Mobile (HangZhou) Information Technology Co.,Ltd., Hangzhou, China
cszyingp@yahoo.com
[3] School of Computer Science and Technology, Shandong University, Jinan, China
guoshanqing@sdu.edu.cn

Abstract. Deep Packet Inspection (DPI) is widely used in network systems and the processing speed of DPI is very critical. 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 method called Delayed Signature Matching (DSM), which could greatly reduce the number of matching attempts. In the method we do not always immediately match received packets to the signatures, but instead we predefine some protocol specific rules, and evaluate the packets against these rules first to decide when to start signature matching and which signatures to match, thus eliminate lots of useless matching attempts. The proposed DSM method is very suitable for the network auditing scenario since recognizing a flow at the earliest possible time is not required, and the potential seconds of delay brought in by DSM is acceptable. We also find that in the DSM method the number of matching attempts for a flow is unrelated to the number of supported protocols, which is a good property since the number of supported protocols keeps growing. 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 27%∼40%.

Keywords: DPI · Deep Packet Inspection · Delayed Signature Matching · DSM · Fast Path.

1 Introduction

Deep Packet Inspection (DPI) is integrated into many network system today [22, 1, 7, 8, 11, 6]. For example, DPI is used in Firewalls [22], network security monitors [13], and intrusion detection systems (IDS) to recognize the protocols of packets for checking further threats in the application layer. Network auditing systems, which may be required by government regulations (e.g., for monitoring public Wi-Fi services), or by the companies (e.g., for monitoring employees' Internet accessing), also use DPI to recognize which websites users are visiting and which applications users are using.

The processing speed of DPI is quite critical, since one DPI instance usually needs to process traffics from many different terminals, and the volume of the data usually is very big. Thus, the performance is an important consideration for DPI products. For example, commercial products like Qosmos [4] and PACE [5] claim to handle up to 9∼10 Gbps per core, and open source solution like nDPI could handle up to 8.85 Gbps per core as well [1]. Improving the DPI performance could enable the same cores to support more traffics[6], or reduce the number of needed CPU cores. Since the core work of existing DPI is to match received packets against known signatures (or called patterns), many researchers tried to improve the signature matching algorithms, e.g., by modifying the construction of deterministic finite automaton (DFA) [11, 7].

In this paper, we propose a new method focusing on the scheduling of signature matching. By looking closely at existing signature matching process, we find some signature matching attempts are wasted since the needed packets have not been received yet. So we propose a method called Delayed Signature Matching (DSM). In the method, we predefine some rules for targeted protocols, and evaluate the received packets against these rules. If the packets pass the rules, we could start signature matching with the signatures defined by the rules. If the packets do not pass the rules eventually, we will use the original processing method as usual. Intuitively, these rules produce *fast paths* in the signature matching (quickly find and match against the proper signatures). We analyze the correctness and performance of the DSM method. We also find an interesting property of DSM: the number of signature matching attempts needed for a flow is constant, and does not grow with the number of supported protocols as the original method does. The delay with DSM is only several seconds at most, which is certainly acceptable in the network auditing scenario. We implement a DSM prototype supporting HTTPS, HTTP, FTP, and POP3 protocols in the open source DPI library nDPI [1, 2], and evaluate it with different datasets. We find that with DSM support for only 4 protocols, the prototype has 27%∼40% performance boost in the signature matching.

2 Related Work

There are many existing DPI systems [3, 4, 1, 13, 17]. The performance comparisons among some of them can be found in [6]. The DPI systems can be roughly classified [17] into regex-only [3, 4, 15] and hybrid [1, 13, 17] (i.e., combining regex and code) types. L7 filter [3] is a typical regex-only DPI system, which contains many regexes defined for different protocols, and a protocol's detection mainly relies on its regex (unfortunately, the regexes have not been updated since 2009). nDPI [1, 2] is a hybrid DPI system, and is forked from OpenDPI, which is an

[4] https://qosmos.com/

[5] https://ipoque.com/products/dpi-engine-rsrpace-2

[6] For example, the mobile subscribers of China Mobile Inc. consumed 23% more traffics in 2018 Q1, comparing with 2017 Q1.

open source classifier derived from early versions of PACE (PACE is a commercial DPI product mentioned before) according to [6]. nDPI mainly uses code to match different protocols; however, it also supports automaton and Hyperscan [4] in some steps like host name match. nDPI is open source in Github and under active development by the ntop company. It can detect 240+ protocols now, and is used in another ntop product nProbe as well. nDPI's guess_protocol_id also acts as a *fast path* to find the correct protocol parser; however, it is based on the ports and the protocol field of IP header of *one packet* only, and can not fully eliminate useless matching attempts as well (we will show that in Section 3).

Many researchers focus on developing new algorithm to speed up the matching of patterns. Kumar et al. proposed *Delayed Input DFA* [11] which substantially reduces space requirements as compared to a DFA. Dharmapurikar et al. proposed to store signatures in bloom filters to implement matching in hardware [12]. Bremler-Barr et al. proposed to use repetitions in flows to skip repeated data by modifying the Aho-Corasick Algorithm [7]. On the other hand, recently, the privacy of deep packet inspection is gaining more attentions [8–10, 16], since detecting patterns in encrypted traffics like HTTPS is demanded [14], and at the same time DPI is increasingly running as a service in the public cloud platforms. These performance or privacy improvement researches usually are orthogonal to the DSM method proposed in the paper, since they focus on the exact matching algorithms, and the DSM method focuses on the *scheduling* of matching.

3 A Motivating Example

We here use an FTP example to describe how the DPI process works and why there are rooms to improve. We use the process of nDPI [1, 2] for example, and other DPI engines like [3] are similar in general.

We show the first 9 packets of a typical FTP connection in Fig. 1. It contains a 3-way TCP handshake, and later USER and PASS commands to authenticate the client "demo".

We then show how the packets of the FTP connection in Fig. 2 are processed by the nDPI engine. Packets are processed in flows in nDPI and all the 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 some parsers later find the flow does not match them at all so they exclude themselves for the flow early. For packet #4, the engine first tries the guessed FTP protocol parser based on port 21; however, the FTP parser still cannot confirm that it is an FTP flow at that time (it needs more packets to confirm). The following packets are still no match for detection, and only more parsers exclude themselves for the flow. Finally, the packet #7 with reply code 331 makes the FTP parser believe 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 FTP detection.

We can see that there are some matching attempts wasted in the processing. For example, matching the packet #4 to the 103 protocol parsers is doomed to

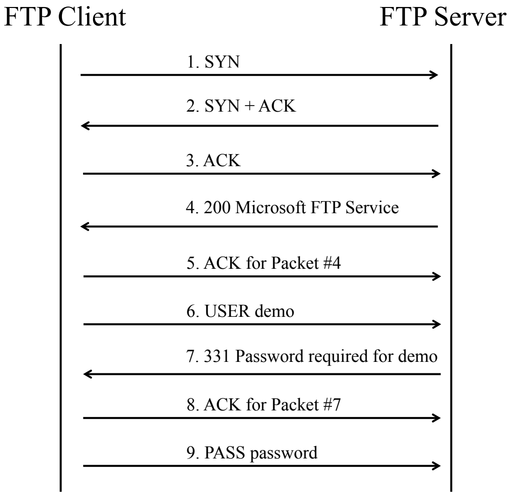

Fig. 1. The initial packets transferred during a typical FTP connection.

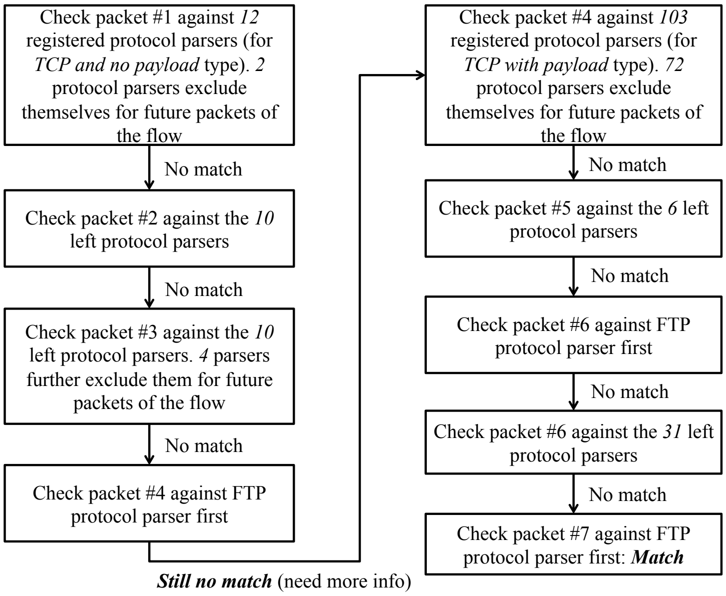
Check packet #1 against *12* registered protocol parsers (for *TCP and no payload* type). *2* protocol parsers exclude themselves for future packets of the flow
No match
Check packet #2 against the *10* left protocol parsers
No match
Check packet #3 against the *10* left protocol parsers. *4* parsers further exclude them for future packets of the flow
No match
Check packet #4 against FTP protocol parser first
Still no match (need more info)
Check packet #4 against *103* registered protocol parsers (for *TCP with payload* type). *72* protocol parsers exclude themselves for future packets of the flow
No match
Check packet #5 against the *6* left protocol parsers
No match
Check packet #6 against FTP protocol parser first
No match
Check packet #6 against the *31* left protocol parsers
No match
Check packet #7 against FTP protocol parser first: ***Match***

Fig. 2. The exact steps of how the packets in previous FTP connection example are processed. 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.

be useless. Though nDPI uses code-based match for FTP protocol, the situation is similar for regex-based DPI engines like L7 Filter and Hyperscan. In L7 Filter the engine keeps appending new packet's content to its flow's received content buffer (2048 bytes at most) and matching the buffer against all protocols' patterns [3]. L7 Filter may use the pattern "ˆ220[\x09-\x0d -∼]*ftp|331[\x09-\x0d -∼]*password" [5] 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 [4] which only needs to feed newly received packet into the library, the whole signature database of all protocols needs to be matched to the newly received packets again and again. Hyperscan also officially advises to avoid big "union" database if possible for performance[7], which makes sense in the regex matching theory [15]. In this paper, we would like to reduce the match searching between the contents and patterns *in essence*. With the DSM method proposed in the paper, the DPI engine only needs to match *each packet against one protocol pattern* 7 times (in contrast to the original 175 times) in total for the FTP connection example.

4 Delayed Signature Matching

The basic idea of Delayed Signature Matching (DSM) is that instead of immediately matching received packets to the signatures, we wait for "enough" packets first. Then the problem is how to determine that currently received packets of a flow are enough. Our solution is to predefine sequences of rules for different protocols, and then evaluate the received packets against the rules first. When the packets pass the sequence of rules of a protocol, we could start signature matching with the protocol's signature. Any failures during the process (e.g., failed to pass the rules) lead to using the original processing method as fallback. Note the rules should be fast to evaluate, and the passing of a sequence of rules should indicate the flow has high probability to match corresponding protocol (otherwise the matching will be wasted). The sequences of rules in effect create *fast paths* in the signature matching, since the packets could directly match against proper signatures so fewer matching attempts are needed.

We show the framework of the delayed signature matching (DSM) method in Algorithm 1. The first flow maintaining step includes finding or creating a flow for the input packet, and updating the variable values used in the *primitives* (introduce later) for the flow. Then for a flow that is not detection completed, the engine checks whether the flow is marked to use the original processing method, or marked to call selected protocol parsers (introduce later) only. In the two cases the engine processes the packet accordingly. For other cases it checks whether the packet could match any DSM rule. If so it will use the instruction of the rule to determine what to do next, like calling the protocol parsers selected

[7] http://intel.github.io/hyperscan/dev-reference/performance.html#unnecessary-databases

by the rule, and waiting for the next packet, etc. At last, if the packet could not match any rule, the engine uses the original processing method as fallback.

We can see the rules are the core part of DSM and in the left part of the section we will focus on introducing them, and use the rules of several protocols as examples.

Algorithm 1 Packet processing with delayed signature matching (DSM).

```
Input: a packet, DSM rules of protocols.
Do flow maintaining: find or create the flow for the packet
if The flow is not detection completed then
    if The flow is marked to use the original processing method then
        Use the original processing method
    else if The flow is marked to call the selected protocol parsers only then
        Call the selected protocol parsers
    else if Match any DSM rule then
        Follow the rule's instruction, which could be:
        - evaluate next rule and follow its instruction
        - call the rule's selected protocol parsers
        - mark the flow to use the original processing method, and process the buffered
          packets and current packet
        - buffer the packet and wait for next packet
        - mark the flow to use currently selected protocol parsers only
    else
        Use the original processing method
    end if
end if
```

First, we introduce a few types of *primitives* that we use in the DSM rules. These primitives are all simple and can be implemented efficiently, which make sure the rules can be checked very quickly.

- *protocol* type comparisons. For example, *protocol* ==TCP.
- *server_port* arithmetic comparisons, including ==, $\geq$, etc. For example, *server_port* == 21.
- *pkt_num* arithmetic comparisons, including ==, $\geq$ etc. *pkt_num* stands for the number of total packets have been received in the flow.
- *payload_pkt_num* arithmetic comparisons. *payload_pkt_num* represents the number of packets that have payload have been received in the flow.
- *tls_pkt_num* arithmetic comparisons. *tls_pkt_num* means the number of total packets that are of Transport Layer Security (TLS) record layer type [19] have been received in the flow. We implement the primitive by simply checking the first byte for 20 (change_cipher_spec), 21 (alert), 22 (handshake), and 23 (application_data) values.
- *payload*[*i*] =='A' equation check. It means to check whether the byte at index *i* of the packet payload is equal to *char* 'A'.

- $payload(i, len)$ ==“ABC” equation check. It means to check whether the *len* bytes start from index i of the packet payload are equal to *string* “ABC”.

Then, we show the processing of FTP and POP3 protocols with DSM rules in Fig. 3. In the flowchart, *the DSM rules are represented by diamonds, and the instructions of rules are represented by rectangular boxes following the rules.* For FTP, after the common flow maintaining process, and assuming the engine needs to check the packet with DSM rules (e.g., the flow is not detection completed etc., as mentioned in the DSM framework previously), the engine checks the rule *protocol* ==TCP first. If the packet passes the rule, it checks another rule on whether the port is the common FTP port 21. If not it will further check other port rules if any. If port matches 21 and it will check whether the $payload\_pkt\_num \geq 4$, which means the packet #7 in Fig. 1 should have been received for the flow. If the rule is not satisfied, another rule $pkt\_num \geq 10$ will be checked, which means whether a threshold number of packets (here we set to 10, same as nDPI) have been received. If *pkt_num* exceeds the threshold, DSM will fall back to the original method, otherwise it will wait for the next packet. If the rule $payload\_pkt\_num \geq 4$ satisfies, it will check the $payload[0]$ against ‘P’, i.e., the first character of the “PASS” command. If the rule satisfies again, then there are enough evidences that it is an FTP flow, so the rule selects the FTP protocol parser to parse all packets received so far. If the result of the parser is matched then we mark the flow as detection completed; the flow successfully goes through the fast path created with DSM rules. Otherwise, it is not an FTP flow and we exclude FTP from the possible protocol list and use the original packet processing method.

The DSM rules of the POP3 protocol are quite similar to the rules of FTP, except that we now match the “USER” command of POP3 [21] 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., $payload\_pkt\_num \geq 2$.

At last, we show the DSM rules we created for HTTPS in Fig. 4. HTTPS is becoming prevalent and more than 30% top 1,000,000 websites use HTTPS by default now (i.e., redirecting HTTP pages to URLs with HTTPS) [8]. Similarly, the *protocol* rule and *server_port* rule are checked first. 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) [19]. If both rules are satisfied then the flow is highly likely to be an SSL connection now, so the HTTPS (TLS) protocol parser is selected. Similarly, if the parser does confirm the application protocol is matched, we mark the flow as detection completed. Otherwise, however, we cannot simply exclude SSL protocol from the flow, because the flow may be of TLS type, but the TLS protocol parser needs more packets to further detect its application protocol. So we add a rule to check whether it is a TLS flow by checking whether the server name has been gotten (either from the Server Name Indication (SNI) extension [20] of TLS, or from the server’s certificate),

[8] https://statoperator.com/research/https-usage-statistics-on-top-websites/

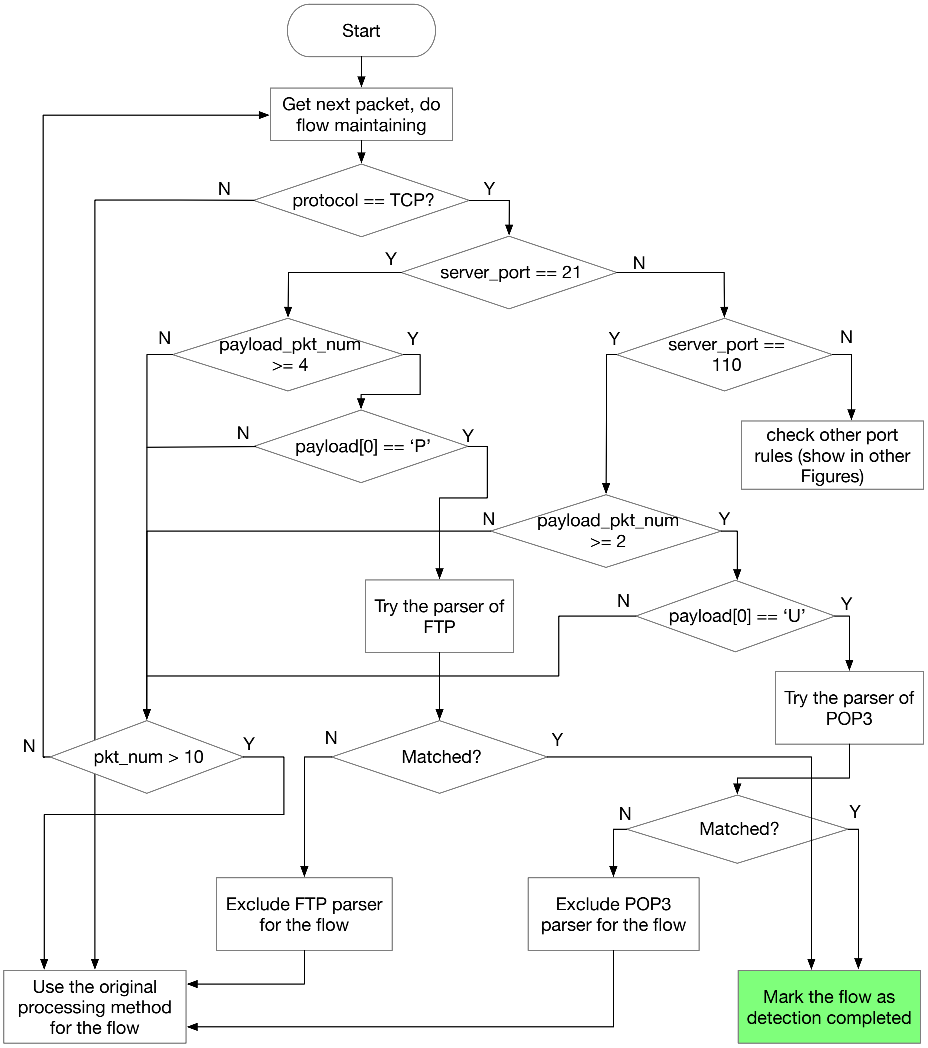
Start
Get next packet, do flow maintaining
N
protocol == TCP?
Y
Y
server_port == 21
N
N
payload_pkt_num >= 4
Y
Y
server_port == 110
N
N
payload[0] == ‘P’
Y
check other port rules (show in other Figures)
N
payload_pkt_num >= 2
Y
Try the parser of FTP
N
payload[0] == ‘U’
Y
Try the parser of POP3
N
pkt_num > 10
Y
N
Matched?
Y
N
Matched?
Y
Exclude FTP parser for the flow
Exclude POP3 parser for the flow
Use the original processing method for the flow
Mark the flow as detection completed

Fig. 3. The DSM rules for FTP and POP3. The rules are represented by diamonds, and the instructions of rules are represented by rectangular boxes following the rules. Flows that could pass through the DSM rules to the downright green box of the flowchart are favored, since they pass through the fast path and have very few matching attempts.

or whether the TLS process has passed some stages (represented by ssl_stage). If we confirm that it is a TLS flow, we will mark the flow to always use the selected TLS protocol parser only. Otherwise, we exclude the TLS protocol from consideration and use the original processing method as well.

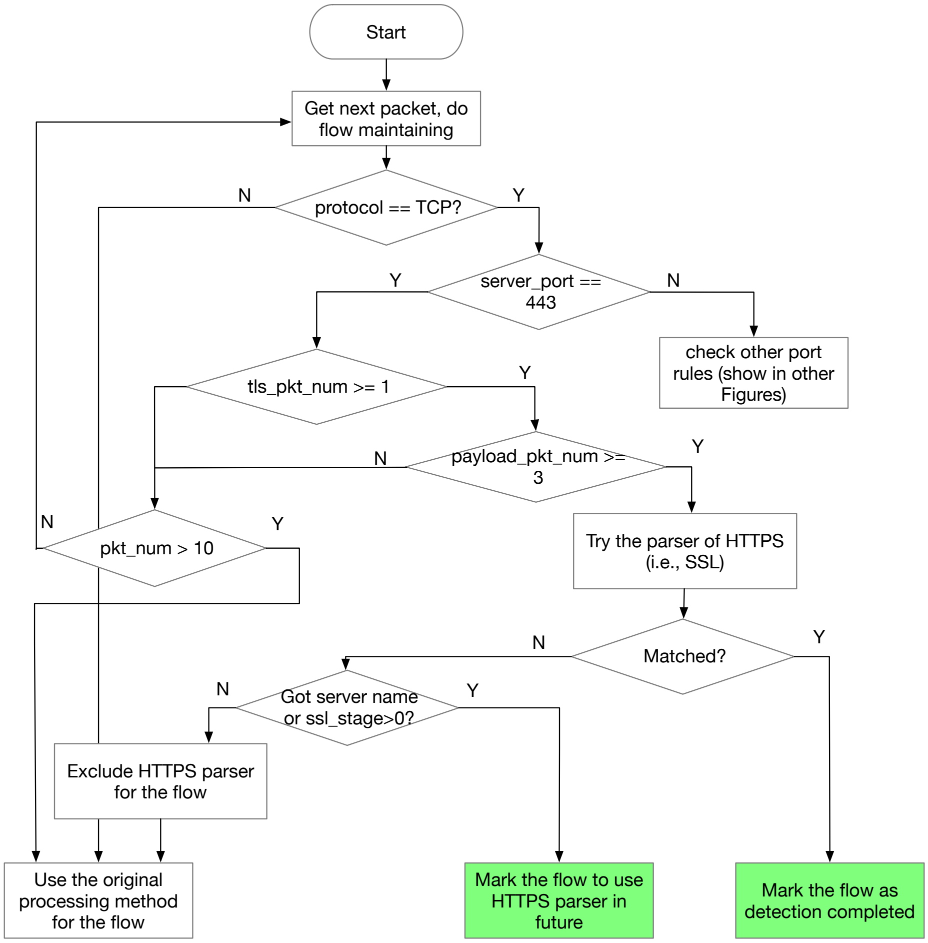
Start
Get next packet, do flow maintaining
protocol == TCP?
N
Y
server_port == 443
Y
N
check other port rules (show in other Figures)
tls_pkt_num >= 1
Y
payload_pkt_num >= 3
N
Y
N
pkt_num > 10
Y
Try the parser of HTTPS (i.e., SSL)
Matched?
N
Y
Got server name or ssl_stage>0?
N
Y
Exclude HTTPS parser for the flow
Use the original processing method for the flow
Mark the flow to use HTTPS parser in future
Mark the flow as detection completed

Fig. 4. The DSM rules for HTTPS. Similarly, flows that could pass DSM rules to the two downright green boxes of the flowchart are favored.

Note that we need to periodically check for flows that applied DSM (i.e., having buffered packets) but got stuck for some time, and use the original processing method to process them. This is because the flows may satisfy some rules of a protocol but cannot pass through them. For example, the *server_port* rule is matched but the *payload_pkt_num* is not enough. On the other side the flows may not have enough packets to eventually satisfy the threshold rule, i.e., the *pkt_num* $\geq$ 10 rule. If we do not use the original processing method to process them, they will not be processed by any parsers even if some parsers can recognize them. We set the stuck time threshold to 2 seconds, which should be sufficient for existing protocols. Also, for efficiency such periodical check for stuck flows could be done together with existing idle flow cleaning process.

Another thing needs to be noted is how we define DSM rules. First, we prefer to wait for a bit more-than-enough number of packets before we begin the signature matching (e.g., for FTP we require that *payload_pkt_num* $\geq$ 4 but not 3). This is because more packets usually imply more evidences on the type of the flow, and thus ensure higher successful probability of the selected protocol parsers. This may not be suitable for the scenarios that require the earliest possible flow recognization like Firewall or IDS [22], but is suitable for network auditing. Second, on the other side, we may intentionally make the rules a bit "relaxed" (e.g., *tls_pkt_num* $>=$ 1 for the TLS protocol) to suit different auditing scenarios and packet transmission strategies, because we want to make the rules more "portable" (written once and suitable for all). For example, in one of our network auditing scenarios, we mainly capture packets of one direction (upstream), and we also find TLS may transmit different record layer packets in one TCP packet. Both the first and second considerations make us prefer to define DSM rules waiting for more packets than the corresponding parser actually needs.

5 Analysis

We give analysis on the correctness and performance of the DSM method below.

5.1 The Correctness of the DSM Method

The delayed signature matching 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. 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 environment. We also confirm that all the protocol parsers we checked in nDPI do have the property.

The second case is that the selected protocol parsers do not recognize the flow's protocol at all, and then we exclude the corresponding protocol 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 based the rules.

The third case is that the selected protocol parsers recognize the flow as their types for sure, but they need more packet to act as detection completed, and we now simply mark the flow to always use the selected protocol parsers in 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 HTTPS, we use the server name and ssl_stage as the indicators of TLS detection.

The fourth case is the selected protocol parsers are still unsure on the protocol of the flow. We didn't show any example on the case previously, since we intentionally 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 apply DSM to these protocols in the first place, or if we do use DSM, we could simply switch to use the original processing method then (but do not exclude the selected protocol parsers), and the correctness is also guaranteed.

We note that the DSM rules could be misled by protocol masquerading mechanisms like FTE [17], since they intentionally change the signatures of protocols to mimic other protocols. Defeating them is out of the scope of DSM 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 [18], DSM could still successfully detect the original protocol.

5.2 Performance Analysis

We analyze the computation cost 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 calls to signature matching 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 ith packet (i starts from 1):

$$S_o = n + a_1 n + a_1 a_2 n + ... + a_1 a_2 ... a_{m-1} n, \quad \text{when } m > 1. \tag{1}$$

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 rules select k protocol parsers. Then we only need at most km calls to signature matching in the two cases (since protocol parsers may even exclude themselves from the detection of the flow later). For other two cases, we at most call signature matching $km + S_o$ times (i.e., DSM fails and the engine uses the original method as fallback). So we could represent the number of calls to signature matching of DSM method as below:

$$S_d = pkm + (1 - p)(km + S_o). \tag{2}$$

Usually we create rules that have high probability to successfully match the flows of the protocol, so p is approximate to 1 (we will later confirm that in Section 6) and then S_d is approximate to km:

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

The value of km usually is much smaller than n (k usually is 1∼2, $m < 10$). 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.

On the other hand, the evaluation of DSM rules does add some costs to the the whole processing. However, we intentionally make the rules simple, and usually only several increment operations and several comparisons are needed. So the added cost is very small, which is also confirmed in our experiments.

We also analyze the extra delay when using DSM. Note that if we create rules that strictly fit to the requirements of the protocol parser (i.e., do not waiting for more packets), and a flow's packets match the rules 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, there are some cases that DSM has extra delay. First, if we create rules that need more packets than the parser actually needs (for the reasons we described before), we will have delay by 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. Second, for the flows that enter DSM process but get stuck there because of no matching rules and not enough packets, the extra delay is related to the stuck threshold time we set (at most 2 times of the threshold). Thus for the 2-second threshold we set, corresponding delay is 4 seconds at most.

6 Evaluation

We implement our DSM method prototype in the popular open source DPI library nDPI [1, 2], with about 600 lines of code. The prototype contains the DSM rules of 4 protocols: HTTPS, FTP, POP3, and HTTP. It is open source for research usage at `https://github.com/zyingp/nDPI/tree/fastpath`[9]. We do all experiments on a PC installing Ubuntu 16.04 LTS, with Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz×4 and 16 GB memory.

We prepare 6 different datasets for the experiments, as shown in Table 1. The datasets come from two testbeds and a web crawler. 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 traffics* (i.e., upstream only, except for several

[9] We also put our raw experiment results there.

special TCP ports) for auditing. One testbed contains 2 China Mobile enterprise WiFi gateways (i.e., hotspots), with 9 computers and phones connected. Another testbed contains 1 WiFi gateway with 2 computers and 1 phone connected (mainly computer traffics). TB_A36, TB_A79, and TB_A304 datasets are from the same testbed A but captured at different times and have different sizes. TB_B326 is from the testbed B. In order to test DSM with other traffic patterns, we also used a crawler running on an iPad to browse Alexa Top 100 and Top 500 websites [10] and got two full traffic traces (i.e., both upstream and downstream traffics) named Top100 and Top500.

Table 1. The properties of the datasets.

Name	Size (MB)	Time (hour)	Num. of flows	Num. of pkts	Traffic description
TB_A36	36.2	30.7	12895	133610	upstream, computer & phone
TB_A79	79.6	23.9	30735	422966	upstream, computer & phone
TB_A304	304.6	103.4	110174	1599281	upstream, computer & phone
TB_B326	326.4	93.2	24970	1383572	upstream, mainly computer
Top100	135.2	0.6	3701	222924	both directions, tablet
Top500	685.3	3.1	37473	1035565	both directions, tablet

We first ensure that the DSM method (i.e., the engine only processes the aforementioned four protocols with DSM rules, and processes other protocols with the original method) detects exact the same protocols as the original method alone for different datasets (e.g., the same 240+ protocols nDPI currently supports). During that we fix several implementation bugs. We also discover a bug of nDPI at that time (in the *ndpi_detection_giveup* function, the *protos* union in *ndpi_flow_struct* is always used as *ssl* type while it sometimes is of *http* type).

We then check the numbers that protocol parsers are called for these datasets, and show the result in Fig. 5. The result is very promising; the DSM method reduces about 42% (for TB_B326 dataset) to 76% (for Top100 dataset) of the calls needed in the original method, with DSM rules only for the four protocols. It is reasonable considering the great reduction of parser called times for protocols like FTP.

Next we test the processing throughput of the whole DPI process. We run the built-in ndpiReader program 15 times for each dataset, discard the first 5 runs whose results may not be stable yet (since the program has large file I/O), and calculate the average of the left 10 runs (we use the same strategy for later experiments as well). We show the result in Fig. 6. When DSM is used, the throughputs all are higher than the original method without DSM: speed up about 20% for the TB_A* datasets, 11% for TB_B326, and about 7% for Top100 and Top500 datasets. The improvement is not as much as the previous

[10] https://www.alexa.com/topsites

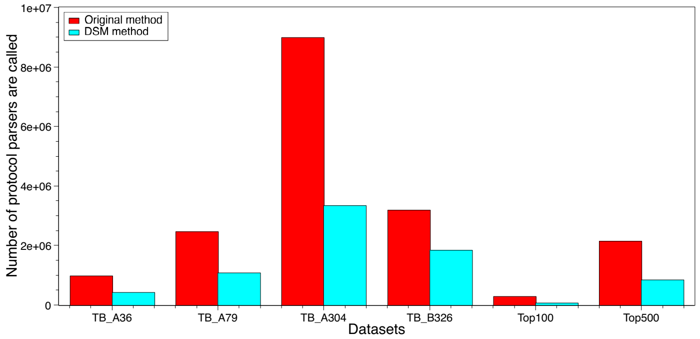

Fig. 5. Comparison on the numbers of the protocol parsers are called. The DSM method omits about 42%∼76% of the calls needed in the original method.

experiment. This is because here the program needs to do more work. The whole process includes loading packets from dataset files, packet unpacking, flow maintaining, and signature matching. We only improve the signature matching part. The other parts amortize the improvement on signature matching, especially for datasets like Top100 and Top500 which have more packets/traffics in a flow so signature matching consumes relatively less time in the whole process.

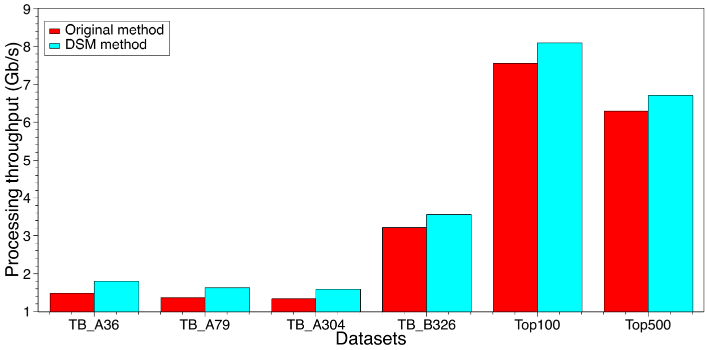

Fig. 6. Processing throughputs of the DSM method and the original method (loading packets from files needs to be done at the same time). DSM improves over the original method about 20% for the TB_A* datasets, 11% for TB_B326, and about 7% for the Top100 and Top500 datasets.

In order to get more accurate results on how the DSM method improves the DPI process, we evaluate it in refined scopes. First, we compare only the time spent on signature matching, which includes basic packet analysis (e.g., processing tcp flags), calling protocol parsers, and also rule evaluation for DSM. The result is shown in Fig. 7. We can see DSM could improve over the original method about 27%∼40%. Then, we check the DSM's improvement on the whole packet processing (including all the work the program does in Fig. 6 except loading packets from files). This is more interesting since it contains actually all the work after we get a packet, either from pcap file or live capture (via libpcap, DPDK, or PF_RING). We show the result in Fig. 8. We can see that the DSM method consumes only 78%∼90% of the time of the original method (improving 10%∼22%).

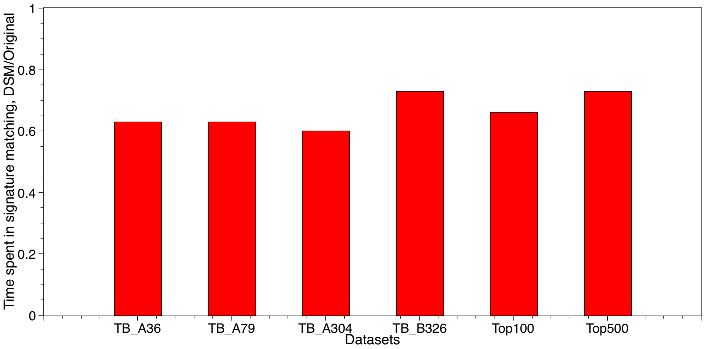

Fig. 7. The ratio of the DSM method to the original method on the time spent only on signature matching (and including rule evaluation for DSM). DSM improves over the original method about 27%∼40%.

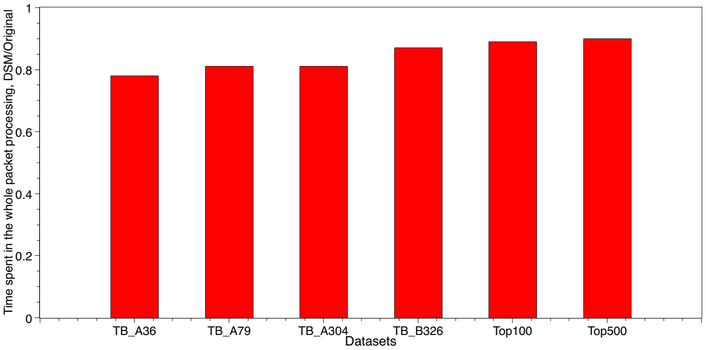

Fig. 8. The ratio of the DSM method to the original method on the time spent on whole packet processing (excluding the time spent on loading packet from files, but including the time on all other work like packet unpacking, flow maintaining, and signature matching). DSM improves over the original method about 10%∼22%.

Finally, we would like to check the effectiveness of current DSM rules. We calculate the number of flows that try the DSM selected protocol parsers (i.e., for the coverage ratio), and we also calculate the number of flows that try the DSM selected protocol parsers but fail to complete detection and turn to the original method as fallback (i.e., for the fail ratio/successful ratio). We show the result in Fig. 9. We can see that the coverage of our DSM rules is 30%∼60% for these datasets, which is promising since we only implement rules for four protocols. Most of the coverage is due to the DSM rules of two protocols: HTTPS and HTTP. Also, the flows tried DSM but failed have a very small proportion, only 0.2%∼2%. We also look into the fail cases and find the majority is due to the wrong protocol exclusion implementation of the nDPI HTTP protocol parser (e.g., a retransmission of a partial HTTP request may not have a line structure and the HTTP protocol parser will wrongly conclude that it is not HTTP).

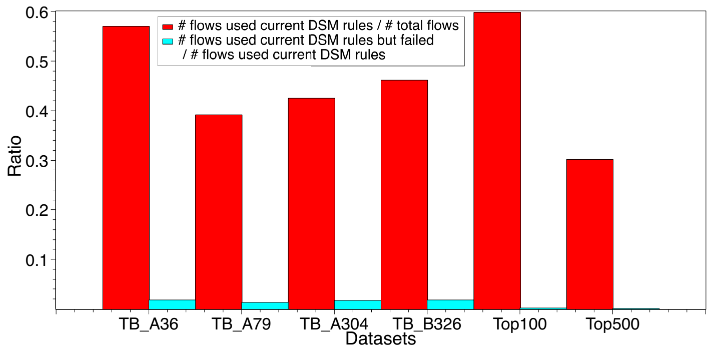

Fig. 9. The flow coverage and successful ratio of the DSM rules in our prototype. The flow coverage of current DSM rules is 30%∼60%. The ratio of flows tried DSM but failed (still fell back to the original method) to all flows tried DSM is only 0.2% to 2%.

7 Conclusions

In this paper we propose delayed signature matching (DSM) for reducing useless signature matching attempts. We achieve that by defining rules to tell when the signature matching could start for a flow and which protocol parsers to use. If a flow does not match any rules then the original method is used. We analyze the DSM method to show its correctness and efficiency. We also show the delay caused by DSM is at most several seconds which is affordable in network auditing, and may also be affordable in other scenarios that do not need real-time actions to packets. We implement DSM rules for four protocols including HTTPS in the open source DPI library nDPI, and evaluate them with different datasets. The result shows that the DSM method accelerates signature matching about 27%∼40%, and accelerates the whole process 7%∼20% (the more the signature matching time accounts for the total time, the more that whole process is accelerated).

References

1. L. Deri, M. Martinelli, T. Bujlow and A. Cardigliano, nDPI: Open-source high-speed deep packet inspection, In Proceedings of International Wireless Communications and Mobile Computing Conference (IWCMC), pp. 617–622. Nicosia (2014)
2. nDPI, `https://github.com/ntop/nDPI`. Last accessed 5 Jun 2018
3. L7-filter, `http://l7-filter.sourceforge.net`. Last accessed 5 Jun 2018
4. Hyperscan, `http://intel.github.io/hyperscan`. Last accessed 5 Jun 2018
5. A. Bober, Introduction to Layer 7-filter, `https://mum.mikrotik.com//presentations/PL10/l7_interprojekt.pdf`.
6. T. Bujlow, V. Carela-Español, P. Barlet-Ros, Independent comparison of popular DPI tools for traffic classification, Computer Networks **76**, 75–89 (2015)
7. B. Anat, T. Shimrit, H. Yotam, H. David. Leveraging traffic repetitions for high-speed deep packet inspection, In Proceedings of INFOCOM, pp. 2578–2586. (2015)
8. J. Sherry, C. Lan, R. Ada Popa, and S. Ratnasamy. 2015. BlindBox: Deep Packet Inspection over Encrypted Traffic. In Proceedings of SIGCOMM, pp. 213–226 (2015)
9. X. Yuan, X. Wang, J. Lin and C. Wang, Privacy-preserving deep packet inspection in outsourced middleboxes, In Proceedings of INFOCOM 2016, pp. 1–9. San Francisco, CA (2016)
10. L. Schiff and S. Schmid, PRI: Privacy Preserving Inspection of Encrypted Network Traffic, In Proceedings of IEEE Security and Privacy Workshops (SPW), pp. 296–303. San Jose, CA (2016)
11. Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, Jonathan Turner, Algorithms to accelerate multiple regular expressions matching for deep packet inspection, In Proceedings of SIGCOMM, pp. 339–350, Pisa, Italy (2006)
12. S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull and J. W. Lockwood, ”Deep packet inspection using parallel bloom filters,” IEEE Micro **24**(1), 52–61 (2004)
13. V. Paxson. Bro: a system for detecting network intruders in real-time. In Proceedings of the 7th conference on USENIX Security Symposium (1998)
14. Z. Durumeric, Z. Ma, D. Springall, R. Barnes N. Sullivan, E. Bursztein, M. Bailey, J Alex Halderman , V. Paxson, The Security Impact of HTTPS Interception, In Proceedings of NDSS (2017)
15. A. Backurs, P. Indyk, Which Regular Expression Patterns are Hard to Match?, In Proceedings of FOCS (2016)
16. R. Poddar, C. Lan, R. Ada Popa, S. Ratnasamy, SafeBricks: Shielding Network Functions in the Cloud, In Proceedings of NSDI (2018)
17. K. Dyer, S. Coull, T. Ristenpart, and T. Shrimpton, Protocol Misidentification Made Easy with Format-Transforming Encryption. In Proceedings of CCS (2013)
18. L. Wang, K. P. Dyer, A. Akella, T. Ristenpart, and T. Shrimpton. Seeing through Network-Protocol Obfuscation. In Proceedings of CCS (2015)
19. T. Dierks, and E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.2, RFC 5246, August 2008
20. D. Eastlake, Transport Layer Security (TLS) Extensions: Extension Definitions, RFC 6066, January 2011
21. J. Myers. and M. Rose, Post Office Protocol - Version 3, RFC 1939, May 1996
22. T. Porter, The perils of deep packet inspection, Security Focus, 2005