TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 13/15 pp422-431
Volume 16, Number 4, August 2011

Accelerating Application Identification with Two-Stage Matching and Pre-Classification*

HE Fei (何 飞)[1,2], XIANG Fan (项 帆)[1], SHAO Yiyang (邵熠阳)[1], XUE Yibo (薛一波)[2,3], LI Jun (李 军)[2,3,**]

1. Department of Automation, Tsinghua University, Beijing 100084, China;
2. Research Institute of Information Technology, Tsinghua University, Beijing 100084, China;
3. Tsinghua National Lab for Information Science and Technology, Beijing 100084, China

Abstract: Modern datacenter and enterprise networks require application identification to enable granular traffic control that either improves data transfer rates or ensures network security. Providing application visibility as a core network function is challenging due to its performance requirements, including high throughput, low memory usage, and high identification accuracy. This paper presents a payload-based application identification method using a signature matching engine utilizing characteristics of the application identification. The solution uses two-stage matching and pre-classification to simultaneously improve the throughput and reduce the memory. Compared to a state-of-the-art common regular expression engine, this matching engine achieves 38% memory use reduction and triples the throughput. In addition, the solution is orthogonal to most existing optimization techniques for regular expression matching, which means it can be leveraged to further increase the performance of other matching algorithms.

Key words: application identification; deep inspection; regular expression; traffic classification

Introduction

Modern datacenter and enterprise networks require granular traffic control to either improve data transfer rates or ensure network security. Flow-based policy control and traffic management are being more and more widely used in both datacenter and enterprise networks[1,2]. Application visibility is critical to understanding network risks and to achieving flow-based traffic control in these networks. However, identifying application-layer protocols is more challenging than traditional traffic classification based on packet headers (also called packet classification). Application identification (App-ID) in network devices needs to meet the high throughput requirement of large networks (e.g., 40 Gbps and beyond) and also be able to scale up with emerging application protocols.

The problem of determining application-layer protocols of network traffic is known as application identification or traffic classification at the application layer. Traditionally network devices depend on well-known port numbers to identify applications. Port-based methods were effective since many applications use IANA registered port numbers (for example, HTTP traffic uses TCP port 80 and DNS traffic uses TCP/UDP port 25). However, emerging applications such as audio and video streaming, file sharing, and social networks are capable of using non-standard or dynamic ports, encapsulated inside commonly used protocols as a means of evading port-based identification. Several application identification technologies

Received: 2010-12-23; revised: 2011-06-03

* Supported by the National High-Tech Research and Development (863) Program of China (No. 2007AA01Z468)

** To whom correspondence should be addressed.
E-mail: junl@tsinghua.edu.cn; Tel: 86-10-62796400

have been recently proposed which can be categorized into payload-based methods[3,4] and statistical methods[5-10].

Statistical methods usually use machine learning techniques to classify the traffic based on flow statistics such as duration, mean packet size or inter-arrival time. However, statistical methods are known to have limitations which make them not suitable for practical usage in production systems. First, statistical methods are currently not able to provide fine-grained application identification. Second, the accuracy of statistical methods is usually not acceptable for in-line network devices. Moreover, the statistical characteristics of traffic on a link may vary with the link usage which further affects the accuracy of statistical methods. For these reasons, most production App-ID systems, such as PaloAlto[11], Juniper[12], and the open source system L7-filter[13], are based on payload-based methods which provide fine-grained, accurate results.

Payload-based methods are accurate in most cases and are similar to other deep inspection (DI) systems, such as intrusion detection systems, which inspect packet payload to search against a set of signatures. However, these methods require more resources (both computational and memory) since every byte of several packets at the beginning of an application session need to be inspected. Nowadays, these signatures are usually defined by regular expressions (regexes) for their expressiveness[13-15]. A rich set of studies[16-19] has been published that give methods to optimize regular expression matching. Unfortunately, few solutions can keep up with the increasing data rates while matching every packet against hundreds of regular expression signatures.

To provide application visibility as the core function of modern networks, the application identification system should meet the following requirements:

(1) High throughput: The application identification system must support wire-speed processing for fine-grained control or traffic management.

(2) Low memory usage: For high speed application identification, the data structures need to be deployed on fast memory (e.g., SRAMs) that has limited capacity due to the high price. Therefore, the memory usage should be as low as possible.

(3) Identification accuracy: Application identification must provide both low false-negative and low false-positive rates.

This paper focuses on payload-based methods that are more accurate, and proposing a matching engine designed to meet both the requirements of high throughput and low memory usage.

Existing research has mostly focused on common multiple regular expression matching. However, for regular expression matching in App-ID systems, if the App-ID characteristics are taken into account, high-level optimizations become possible. The paper uses the App-ID characteristics to divide the problem space and optimize the expected (common) case. The proposed two-stage matching engine is able to split the signature set into smaller sets, and each flow only needs to match one or two of them. Most existing regex matching algorithms can be used to further increase the performance of the sub-matching procedure in this solution.

The main contributions of this study include:

(1) A two-stage matching engine is proposed to provide much higher throughput than a state-of-the-art multi-DFA matching engine[18]. Evaluations using real world traces show that the algorithm is up to 3 times faster than multi-DFA matching.

(2) Group merging algorithms are proposed to optimize the memory usage of the matching engine. The total memory usage of the algorithm is more than 60% less than that of multi-DFA matching.

(3) A trace-driven application identification system is implemented to support evaluations using real world traces.

1 Payload-Based Application Identification

1.1 Architecture of a typical payload-based application identification system

A common architecture of existing payload-based application identification systems is illustrated in Fig. 1. A received packet is first processed in an IP layer processing module, including IP header parsing, IP defragmentation, and other IP layer processing. After the packet header is parsed, several fields of the packet header (usually 5 tuples, including source address, destination address, transport-layer protocol, source port, and destination port) are used as a flow identifier (flow-id) to lookup in the flow table. If the flow that the packet belongs to has not been classified, the packet is sent to a transport layer processing module for TCP stream reassembly or UDP header checking. Then, the packet payload is submitted to the matching engine to search against application signatures. The signature matching engine is the system bottleneck, which is to be optimized in this paper.

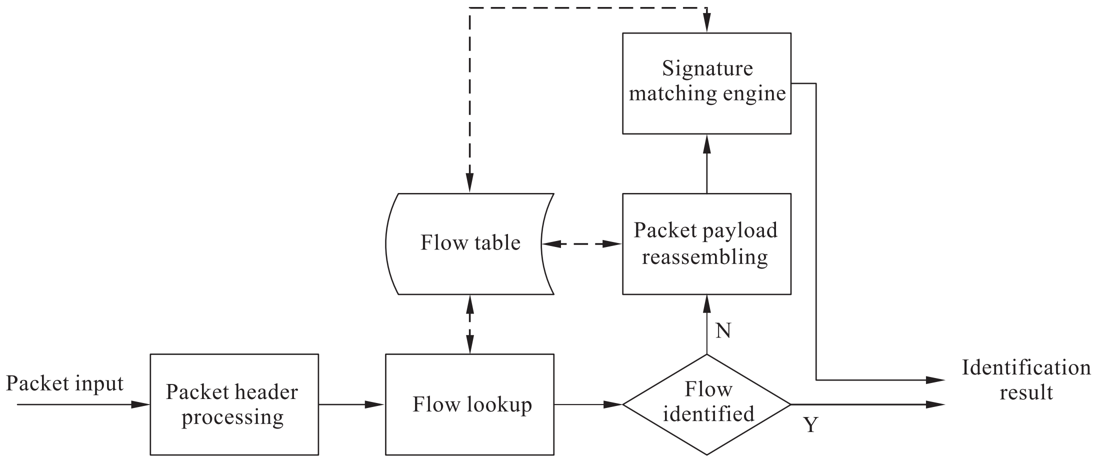

Fig. 1 Typical payload-based application identification system

1.2 Signature matching engine

Regular expressions are widely used as signatures of application identification systems, due to their expressive power and flexibility for describing protocol patterns. For example, all protocol signatures in the L7-filter are written in regular expression. In addition, in the protocol identification module of the Bro intrusion detection system, regular expressions are also used as its pattern language. Thus, the matching engine used in application identification is usually a multi-pattern regular expression matching engine.

High-speed regular expression matching is usually based on finite automata, either deterministic finite automaton (DFA) or nondeterministic finite automaton (NFA). Theoretically, a regular expression of length n can be compiled into an NFA with $O(n)$ states. When an m-state NFA is converted into a DFA, it may generate $O(2^m)$ states in the worst case. However, the processing complexity for each input character is $O(1)$ in a DFA, but is $O(n^2)$ for an NFA. Thus, neither of the standard FA solutions are feasible for regular expression matching engines for high speed packet payload scanning. NFS handles k regular expressions with total length of kn (k is usually hundreds or even thousands and n is the average length of the regular expressions) by either compiling them individually in k automata or into a single automaton. In either case, $O(n)$ states may be active concurrently, which reduces performance with a large number of per-flow states to be maintained. DFA usually can not compile a large signature set into a single composite DFA, since the composite DFA grows exponentially in most cases. Among existing solutions, the multi-DFA approach proposed by Yu et al.[18] that controls the number of DFA states by creating several DFAs based on a grouping approach is the most used practical approach to deal with hundreds of signatures.

2 Two-Stage Matching Engine Framework

2.1 App-ID characteristics

App-ID and IDS are the two most widely used forms of deep inspection systems. Although the core of DI system is a regular expression matching engine, they have some essential differences. The first essential difference is the matching rate. When processing real traffic, the IDS system usually finds very few matches. For example, Sourdis et al.[20] found that over 90 percent of traffic did not match any IDS signature. On the contrary, almost all the flows match certain signatures in an App-ID system. The tests on several real traces (the detailed information is described in Section 4) shown in Fig. 2 illustrate that about 90% of the flows match signatures in the L7-filter.

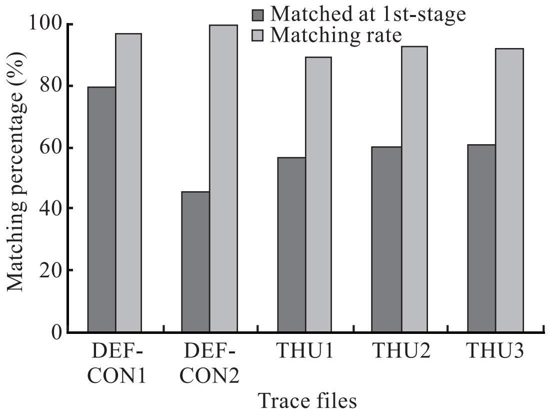

Fig. 2 Matching rate

The second difference is that App-ID signatures have the characteristic that most signatures are anchored, which means the signatures should be matched only at the start of a flow. Figure 3 shows that around 80% of the signatures in the L7-filter are anchored. A detailed study of the most popular application-layer protocols showed that this signature set characteristic is reasonable. The four classes of Internet application-layer protocols are struct-style binary protocols, IETF-style protocols, structured binary protocols, and structured text protocols. Most Internet protocols belong to these four classes, and the parsers of these kinds of protocols need to parse the packets from the beginning of a flow. Therefore, the majority of L7-filter signatures are anchored.

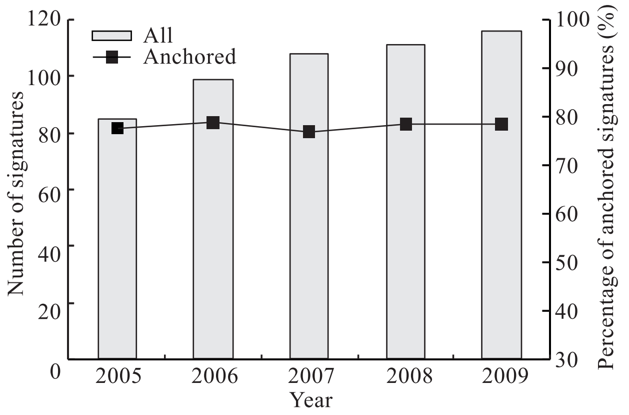

Fig. 3 L7-filter signatures from 2005-2009

2.2 Two-stage matching engine design

An algorithmic solution was then developed to overcome the large overhead faced by DI matching engines. The objective of this solution is to reduce the average number of signatures that packets need to match against.

Since most packets match certain signatures, the pre-filter technology widely used in IDS[20,21] is not suitable for App-ID. To optimize the matching procedure, as much traffic as possible should be matched using as few resources as possible. Our choice is to split the matching procedure into two stages. In the first stage, anchored signatures that can be matched using fewer resources than non-anchored signatures are processed, while non-anchored signatures are processed in the second stage. Figure 2 shows that 60%-80% of real-life traffic can be matched at the first stage.

Figure 4 gives an outline of the proposed solution. The reassembled payload of the packets in a flow is first pre-classified based on *m* bytes of its prefix and assigned a group-id. The purpose of the pre-classifier is to reduce the signatures that a specific payload needs to match against, so that the group of signatures can be compiled into one DFA. Then, the reassembled payload is matched against the DFA specified by its group-id. For the majority of flows, a match is found in this stage, and no further matching is needed. If the payload does not match, it will be sent to the next stage to match against non-anchored signatures.

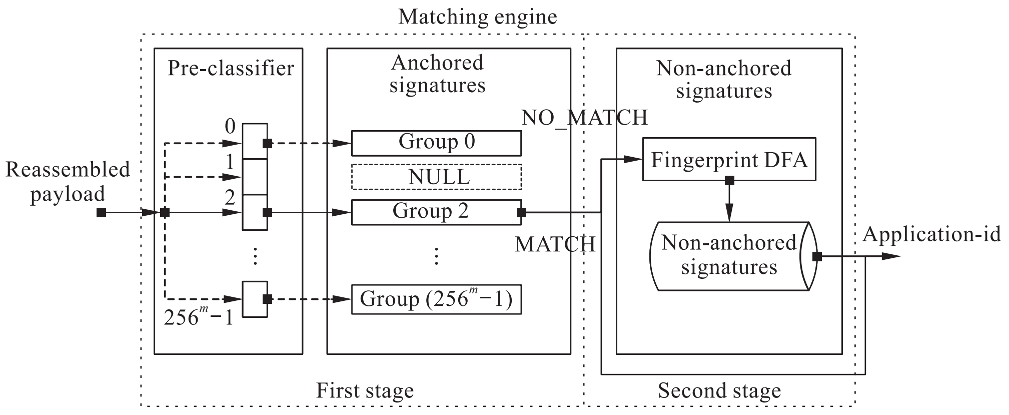

Fig. 4 Two-stage matching engine

3 Detailed Design and Optimization

3.1 Prefix-based classification

Anchored signatures should match from the start of the payload, which means that if the starting characters of a flow are not accepted by the signatures, then there is no need to match these signatures. Therefore, the prefixes of the signatures can be used to pre-classify the flows and narrow the number of signatures that need to be matched.

The main idea is to separate the signature set into exclusive groups based on the first *m* bytes that signatures match. When the reassembled payload of a flow enters the matching engine, it is pre-classified based on its first *m* bytes and only needs to match against a small group of signatures which accept the prefix. Since the anchored signatures are divided into exclusive subsets, each flow only needs to be matched against one subset. Moreover, each subset of anchored signatures can be compiled into a DFA since the number of signatures in a subset is much smaller than that of the entire signature set.

To construct these exclusive subsets, the prefixes of anchored signatures should first be extracted. A regular expression consists of four types of components, exact characters, character classes, wildcards (e.g., dot), and repetitions. We define the prefix length of an anchored regular expression as the largest number of non-wildcard characters from the beginning. For example, the prefix length of regex "^[ab]c.*d" is 2. For a regex that have a prefix length of *k*, up to *k* characters from the beginning of the regex can be used to classify the payloads.

The width of the pre-classifier is defined as the bytes used to group the signatures into subsets. The width of the pre-classifier usually should be the minimum prefix length of all anchored signatures. If it is longer than the minimum prefix length, the signatures having a shorter prefix length will be duplicated in many subsets. Figure 5 shows the distribution of signatures with different prefix lengths in an L7-filter. The minimum prefix length is one, so we set the width of the pre-classifier to one byte. If the width of the pre-classifier is set to a number larger than one byte, the signatures are divided into more fine-grained groups, but there is more redundancy between these groups, since every signature with prefix-length one is duplicated.

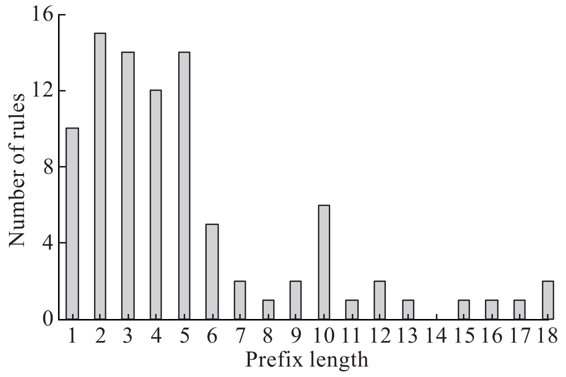

Fig. 5 Prefix length vs. number of rules

3.2 Group merging

The prefix-based pre-classifier divides *m* signatures into several exclusive small groups. The computation complexity for processing the signatures is then reduced from $O(k)$ (k is the number of DFAs in a multi-DFA solution) to $O(1)$. However, the overall number of signatures increases after the signatures are divided into subsets based on their prefixes. When the pre-classifier width is one byte, the total number of signatures increases by 160. Character classes in the prefixes cause duplication of the signatures, which means that some signatures are duplicated several times in different groups. For example, the regex "^[ab]c.*d" should be placed in both the group of prefix "a" and the group of prefix "b".

Group merging is then used to reduce the signature duplication. We first provide a formal definition of the problem: There are *k* groups of signatures. Signatures in these groups are selected from a set of *m* regular expressions based on the prefixes of the regular expressions. Several groups can be merged into one, which means these groups share one DFA. The group-merging problem discussed in this section is to find an optimal combination of *k* groups that results in the smallest memory usage of all the DFAs compiled from the merged groups. $S(\mathrm{A})$ is used to denote the size of the DFA compiled from signature group A. Then, the gain of merging two groups, say group A and group B, is $S(\mathrm{A})+S(\mathrm{B})-S(\mathrm{AB})$. In general, this problem is an NP-hard problem. Although the number of groups is not large, the calculation of the gain of the merging two groups, i.e., generating the DFAs of group A, group B, and the merged group AB, may cost too much time. Therefore, two heuristic algorithms were developed to provide fairly good results.

Merging of two groups may have two opposite effects. On the one hand, the redundancy of same signatures in two groups is removed, but on the other hand, signatures that do not exist in both groups may cause the number of states in the DFA of the merged group to grow exponentially. Following the discussion in Ref. [18], if there are x wildcards per regex, adding one more regex into the DFA increases its size by $(x+1)$ times. For example, adding one regex into the group that consists of regexs with one wildcard on average doubles the size of the DFA, thus the gain of merging is negative. Therefore, a positive merging gain can be found only if the number of different signatures in the two groups is not very large. The interaction between two groups is defined as the number of signatures in the smaller group that do not exist in the larger one, which will be used as a heuristic.

Step 1 Subset merging

Two groups assigned in the pre-classifier can share one DFA if all the signatures in both groups are compiled into a single DFA. Through analysis of the groups of signatures divided based on the prefixes of signatures, we find that many groups are similar. Even more, some groups are subsets of other groups, which means the interaction is zero. Since subset merging only reduces the total size of all DFAs (actually the DFA of the subset is removed), the procedure of subset merging is quite simple. The pseudo-code of the subset-merging algorithm is provided below. Initially, the list of groups G contains the group of signatures divided based on prefix, and the set of group-ids (denoted by gids) contains only one gid. The pseudo-code of the subset-merging algorithm is presented in Fig. 6.

Algorithm 1 Subset merging

```
procedure subset_merging (list G= (G_i (set gids, set sigs)))
    for $G_i \in G$ do
        for $G_j \in G$ && $j > i$ do
            if interaction($G_i$, $G_j$) == 0 then
                group_merge($G_i$, $G_j$);
                move(G.end(), $j$);
end
procedure group_merge (group $G_1$, group $G_2$)
    $G_1$.gids.add($G_2$.gid);
    $G_1$.sigs.add($G_2$.sigs);
End
```

Fig. 6 Subset-merging algorithm

Step 2 Selective merging

After subset merging, it is more challenging to perform further merging. To determine the gain of merging two groups, the composite DFA of two groups and the merged group need to be compiled, which is a time-consuming job. We rely on two heuristics to reduce the search space of the optimal solution. Firstly, according to the analysis above, only the merging of two groups with small interaction may get a positive gain of merging. Therefore, we calculate the interaction of two groups in the selective merging stage, and keep on searching only if the interaction is smaller than a given threshold. Secondly, as we know, even the interaction of two groups is only one or two, the composite DFA of the merged group may grow exponentially if the average number of wildcards in the regexes of the groups is over one (since there are no counting constraints in L7-filters signatures, we only discuss the wildcards here). Based on this observation, we inspect the average number of wildcards of the merged groups before really compiling the composite DFA. If the average number of wildcards is over one, this combination of groups is skipped.

In addition to the heuristic-based searching, we also avoid the recalculating by recording the sizes of every DFA complied, in order to further reduce the overhead of calculating the gain of merging.

The pseudo-code of the selective-merging algorithm is presented in Fig. 7.

Algorithm 2 Selective merging

```
procedure selective_merging (list $G$= ($G_i$ (set gids, set sigs)))
  map size;
  for $G_i \in G$ do
    for $G_j \in G$ && $j > i$ do
      if interaction($G_i$, $G_j$) < Threshold then
        if avg_wildcards(group_merge($G_i$, $G_j$)) < 1 then
          // lazy compiling
          if size($i$) == 0 then
            size($i$) = DFA($G_i$).size;
          endif
          if size($j$) == 0 then
            size($j$) = DFA($G_j$).size;
          endif
          if size($i$) + size($j$) > DFA($G_{ij}$).size then
            group_merge($G_i$, $G_j$);
            move(G.end(), $j$);
          endif
        endif    //avg_wildcards
      endif  // interaction
end
```

Fig. 7 Selective-merging algorithm

3.3 Fingerprint-based matching

As discussed in Section 2, only a little fraction of traffic needs to be processed in the non-anchored stage, and the number of non-anchored signatures is also small. But we still need a scalable solution to deal with these regular expressions. In this paper, we employ a pre-filtering technique similar to the one used in Snort[20], called fingerprint-based matching, to process the non-anchored signatures. The fingerprint-based matching performs multi-string matching on subparts of the signatures, i.e. fingerprints. When a fingerprint is matched, a single regular expression matching, either DFA-based or NFA-based, is invoked.

For a set of non-anchored signatures, we first extract the fingerprints of every signature. The fingerprint of a regex is several fixed sub-strings of the regex. For regex like "first|second.*third", the fingerprint may be "first, second" or "first, third". The procedure of the fingerprint extraction is as follows: Regexes are first split at " | " into different parts, and then all fixed sub-strings in each part are extracted. Finally, the fingerprints of the regexes are selected from these sub-strings to meet two criteria: (1) the fingerprint for each signature is unique; (2) if there are multiple sub-strings for each part of a regex, then the longest one is selected to improve the efficiency.

After all the fingerprints are extracted, we construct an Aho-Corasick automaton[22] to perform fingerprint-lookup. When the reassembled payload matches a certain fingerprint, it is checked against the corresponding regular expression to confirm the match.

4 Performance Evaluation

In this section, we evaluate the effectiveness of our two-stage matching engine by comparing both the memory usage and speed of our solution against a multi-DFA-based matching engine. As discussed in Section 1, multi-DFA-based solution is the best speed and memory usage tradeoff in existing solutions. Compared to the popular multi-DFA-based matching engine, our matching engine consumes 62% to 86% less memory. Experiments that use traffic traces from DEFCON and Tsinghua University networks demonstrate that our solution increases the throughput of the App-ID system up to 3 times.

4.1 Experimental setup

We perform all the experiments using two sets of real world traffic traces. The first set (DEF1, DEF2) is from the Defcon 9 Capture the Flag contest[23], which contains a large amount of anomalous traffic. The second set (THU1-THU3) is from a local LAN with about 1000 computers at Tsinghua University. Most packets in THU traces are normal traffic, which is very different from DEF traces. Among all the packets in these traces, only TCP and UDP packets are processed.

We use the Regular Expression Processor[24] to compile DFAs from groups of regular expressions. The code is modified to support generating multi-DFAs in which the size of each DFA is limited by a threshold.

We implement a trace-driven application identification system based on Libnids[25] and L7-filter to test the performance of our solution in a real App-ID system. The system reads packets from traffic traces and reassembles packets using Libnids and then sends packets to the matching engine. The signatures used are from the latest L7-filter released on July 2009.

All the experiments were obtained on a Server with a Xeon E5504 CPU (4 cores at 2.0 GHz) and 4 GB DDR3 memory.

4.2 Memory usage comparison

The first part of Table 1 shows the results of compiling all L7-filter signatures into multiple DFAs. We set the limit of the number of states in each composite DFA from 10 000 first. The multi-DFA algorithm generates 11 DFAs and the total number of states is 32 715. We do not set the limit to a smaller number, because the algorithm creates more DFAs in that situation, which results in worse matching performance. When we increase the limit to larger numbers, the algorithm creates few groups but the total number of states increases. The number of signature groups can only decrease to 5 for this L7-filter signature set. When we set the limit to 500 000 states, the number of DFAs created is still 5 and the time of compilation needed is about 10 h. For convenience, these multi-DFA settings are denoted as multi-DFA_5, Multi-DFA_6, Multi-DFA_8, and Multi-DFA_11.

Table 1 Memory usage comparison

Algorithm	Composite DFA state limit	Number of DFAs	Total number of states
Multi-DFA	10 000	11	32 715
	20 000	8	54 979
	30 000	6	60 699
	80 000	5	90 802
Two-stage	Initial	62	34 016
	After subset merging	40	25 975
	After selective merging	21	11 448

For our solution, the signature set is divided into 62 groups initially, based on the prefix. The total number of states for these 62 groups is 34 016, which is only a little larger than Multi-DFA_11. We apply our group-merging algorithm to the prefix-based groups. After performing the subset merging, the number of groups is reduced to 40 and the total number of states is reduced to 25 975 correspondingly. The selective merging algorithm is performed to further reduce the memory usage. This heuristic algorithm merges the 40 groups into 21 and successfully reduces the total number of states to 11 448, almost 67% reduction of memory usage.

Figure 8 shows the total memory usage of our two-stage matching scheme and the multi-DFA scheme. For the multi-DFA scheme, the total memory usage includes the memory usage of all the composite DFAs. The memory usage of the two-stage scheme consists of the memory consumption of all the DFAs in the first stage, the fingerprint DFA in the second stage, and the DFAs for every non-anchored signature. For the L7-filter signature set, the fingerprint DFA consumes 365 KB, while the DFAs for each non-anchored signature use 702 KB. From Fig. 8 we can see that the memory usage of the proposed two-stage scheme is only 38.3% of Multi-DFA_11, and 13.8% of Multi-DFA_5.

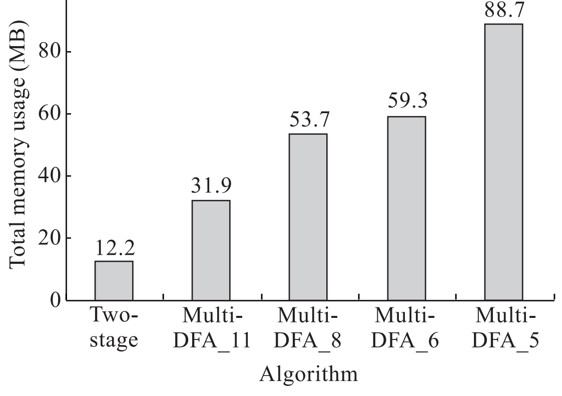

Fig. 8 Total memory usage

4.3 Throughput comparison

Since the difference in the characteristic of traffic may affect the performance of the algorithms, we test the performance of these two algorithms in our trace-driven application identification system using real world traffic traces. Figure 9 shows the average number of memory accesses per input character to the matching engine. The average number of memory accesses of the two-stage matching is from 2.2 to 2.5 for different traffic traces. In comparison, the multi-DFA scheme needs 5-11 memory accesses (2-5 times) per input characters.

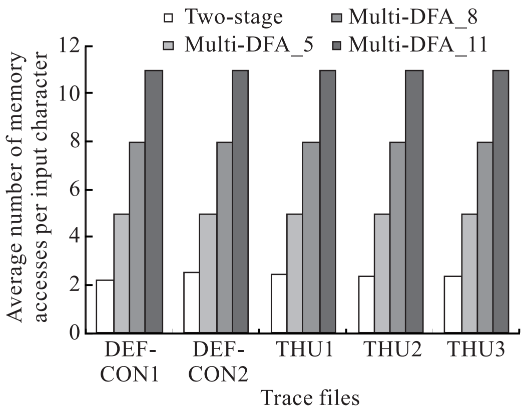

Fig. 9 Memory access comparison

We measure the throughput of these two algorithms by recording the time needed to process the trace files. When calculating the throughput, the time for reading a trace file from the disk is deducted from the processing time. Figure 10 shows the throughput normalized to that of the Multi-DFA_11. Comparing to Multi-DFA_5,

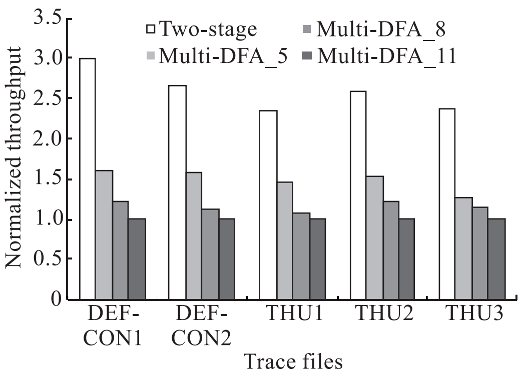

Fig. 10 Throughput comparison

the two-stage matching algorithm yields a performance improvement of 60% to 89%. Also, our algorithm is 2.3 to 3.0 times faster than the Multi-DFA_11. The differences in the performance improvement are mainly affected by the percentage of traffic matched in the first stage.

5 Conclusions

Application identification in network devices is challenging due to its performance requirements including high throughput, low memory usage, and identification accuracy. Existing work on payload-based application identification mainly focuses on optimizing common regular expression matching. They do not utilize the characteristics of application identification that are different from other deep inspection applications. In this paper, we propose a matching engine solution to accelerate application identification by employing two-stage matching and pre-classification techniques. Our solution reduces the memory usage and increases the throughput at the same time. Comparing to the state-of-the-art common regular expression engine, the matching engine achieves up to 38% reduction of memory usage and 3 times throughput increase. In addition, the proposed solution is orthogonal to most existing optimization techniques of regular expression matching, which means we can use these techniques to further increase performance.

References

[1] Al-Fares M, Radhakrishnan S, Raghavan B, et al. Hedera: Dynamic flow scheduling for data center networks. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI). USA: USENIX, 2010.

[2] McKeown N, Anderson T, Balakrishnan H, et al. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Computer Communication Review*, 2008, **38**(2).

[3] Sen S, Spatscheck O, Wang D. Accurate, scalable in-network identification of p2p traffic using application signatures. In: Proceedings of the 13th International Conference on World Wide Web. USA: ACM, 2004: 512-520.

[4] Haffner P, Sen S, Spatscheck O, et al. ACAS: Automated construction of application signatures. In: Proceedings of the 2005 ACM SIGCOMM Workshop on Mining Network Data. USA: ACM, 2005.

[5] Moore A W, Zuev D. Internet traffic classification using Bayesian analysis techniques. In: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. Canada: ACM, 2005.

[6] Callado A, Kelner J, Sadok D, et al. Better network traffic identification through the independent combination of techniques. *Journal of Network and Computer Applications*, 2010, **33**(4): 433-446.

[7] Bernaille L, Teixeira R, Akodkenou I, et al. Traffic classification on the fly. *SIGCOMM Computer Communication Review*, 2006, **36**(2).

[8] Li Zhu, Yuan Ruixi, Guan Xiaohong. Accurate classification of the Internet traffic based on the SVM method. In: Proceedings of IEEE International Conference on Communications (ICC). Scotland: IEEE, 2007: 1373-1378.

[9] Szabo G, Szabo I, Orincsay D. Accurate traffic classification. In: Proceedings of IEEE International Symposium on World of Wireless, Mobile and Multimedia Networks (WoWMoM). Finland: IEEE, 2007: 1-8.

[10] Zhang Guangxing, Xie Gaogang, Yang Jianhua, et al. Accurate online traffic classification with multi-phases identification methodology. In: Proceedings of the 5th IEEE Consumer Communications and Networking Conference (CCNC). USA: IEEE, 2008: 141-146.

[11] Palo Alto Networks Enterprise Firewall. http://www.paloaltonetworks.com/products/pa4000.html. 2011.1.15.

[12] Juniper SRX Services Gateways. http://www.juniper.net/au/en/products-services/security/srx-series/.2010.10.15.

[13] L7-filter. http://l7-filter.sourceforge.net. 2010.10.15.

[14] Cisco Adaptive Security Appliance. http://www.cisco.com.

[15] Bro. http://www.bro-ids.org. 2010.10.15.

[16] Kumar S, Dharmapurikar S, Yu F, et al. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *ACM SIGCOMM Computer Communication Review*, 2006, **36**(4).

[17] Becchi M, Crowley P. An improved algorithm to accelerate regular expression evaluation. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS). USA: ACM, 2007: 145-154.

[18] Yu F, Chen Z, Diao Y, et al. Fast and memory-efficient regular expression matching for deep packet inspection. In: Proceedings of the 2nd ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS). USA: ACM, 2006: 93-102.

[19] Smith R, Estan C, Jha S, et al. Deflating the big bang: Fast and scalable deep packet inspection with extended finite

automata. *ACM SIGCOMM Computer Communication*, 2008, **38**(4).

[20] Snort. http://www.snort.org. 2010.10.15.

[21] Sourdis I, Dimopoulos V, Pnevmatikatos D, et al. Packet pre-filtering for network intrusion detection. In: Proceedings of the 2nd ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS). USA: ACM, 2006: 183-192.

[22] Aho A V, Corasick M J. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 1975, **18**(6): 333-340.

[23] Group S. Defcon 9 Capture the Flag Data. http://ictf.cs.ucsb.edu/data/defcon_ctf_09. 2010.11.1.

[24] Michela Becchi. Regular expression processor. http://regex.wustl.edu/. 2010.11.15.

[25] Libnids. http://libnids.sourceforge.net. 2010.11.15.