Computers, Materials & Continua CMC, vol.63, no.2, pp.649-661, 2020

Skipping Undesired High-Frequency Content to Boost DPI Engine

Likun Liu[1], Jiantao Shi[1, *], Xiangzhan Yu[1], Hongli Zhang[1] and Dongyang Zhan[2]

Abstract: Deep Packet Inspection (DPI) at the core of many monitoring appliances, such as NIDS, NIPS, plays a major role. DPI is beneficial to content providers and censorship to monitor network traffic. However, the surge of network traffic has put tremendous pressure on the performance of DPI. In fact, the sensitive content being monitored is only a minority of network traffic, that is to say, most is undesired. A close look at the network traffic, we found that it contains many undesired high frequency content (UHC) that are not monitored. As everyone knows, the key to improve DPI performance is to skip as many useless characters as possible. Nevertheless, researchers generally study the algorithm of skipping useless characters through sensitive content, ignoring the high-frequency non-sensitive content. To fill this gap, in this literature, we design a model, named Fast AC Model with Skipping (FAMS), to quickly skip UHC while scanning traffic. The model consists of a standard AC automaton, where the input traffic is scanned byte-by-byte, and an additional sub-model, which includes a mapping set and UHC matching model. The mapping set is a bridge between the state node of AC and UHC matching model, while the latter is to select a matching function from hash and fingerprint functions. Our experiments show promising results that we achieve a throughput gain of 1.3-2.6 times the original throughput and 1.1-1.3 times Barr's double path method.

Keywords: Deep packet inspection, pattern matching algorithm, AC.

1 Introduction

In recent years, content-based networks are developing rapidly, such as Software Defined (SDN), Content-Centric Networking (CCN) which bring a new round of development opportunities for content providers. Meanwhile, content providers and censorship are paying more attention to monitor the content of traffic. As for monitoring appliances, common ones include network intrusion detection system (NIDS), network intrusion prevention system (NIPS), spam filtering, network user behavior analysis system, and so on. No matter which one, Deep Packet Inspection (DPI) engine, a prerequisite component, inspects the payload of the packets to detect predefined signatures of malicious information [Afek, Bremler-Barr, Harchol et al. (2016)].

The key technology of DPI is the pattern matching algorithm, which is a mature technology

[1] School of Computer Science and Technology, Harbin Institute of Technology, Harbin, 150001, China.

[2] Department of Computer Science and Engineering, The Ohio State University, Columbus, 43202, USA.

[*] Corresponding Author: Jiantao Shi. Email: shijiantao@hit.edu.cn.

Received: 23 June 2019; Accepted: 16 July 2019.

and has been researched for nearly 40 years. The most famous algorithms are Aho-Corasick algorithm (AC) [Aho and Corasick (1975)], Wu-Member algorithm (WM) [Wu and Manber (1994)] and Set Backward Oracle Matching (SBOM) [Allauzen, Crochemore and Raffinot (1999)]. AC is prefix searching using a trie tree, and it achieves a significant performance boost in short pattern set, while WM is suffix searching using two hash tables, which can skip well-established unmatched characters and perform better than AC in long pattern set. As to SBOM, which is also appropriate for long pattern set, it borrows the factor oracle structure and gets more throughput than WM in same situation. Most of the later algorithms are based on the evolution of these three algorithms [Nelms and Ahamad (2010); Liu, Liu and Tan (2015); Tan, Liu, Bu et al. (2011); Liu, Zhang, Yu et al. (2018); Chen and Wang (2015); Xing and Pao (2018); Yuan, Duan and Cong (2018); Wu, Zhang, Zhang et al. (2018)], especially under a multi-core architecture.

Although these algorithms have made outstanding contributions to improve the performance of DPI, a significant gap remains in skipping extraneous characters. Generally speaking, content providers or censorship are only concerned with a small portion of the payload of the packets. This is verified by actually observing the pattern sets and traffic content of monitoring appliances, and specifically the heavy HTTP traffic. The heavy HTTP traffic clearly displayed a lot of undesired high-frequency content (UHC), including full repetition and partial repetition. The full repetition is the entire string appears many times, such as javascript and stylesheet (e.g., <html, <head>, </style>), while the partial repetition is the substring, such as shared HTML code. What's more, a close look at the heavy traffic and we found that traffic from the same content provider is very similar, for example, the same html framework, similar files. On top of that, since the rapid development of content-based networks (e.g., CDN, SDN), similar content will be routed to the same monitoring appliance. Naturally, such networks provide a hotbed for attackers to evade.

The literature begins by scoping the problem space, and proposes a new matching model. The model is Fast AC Model with Skipping (FAMS), which allows UHC to be skipped rather than scanned again. There are three modules in FAMS: standard AC automaton, mapping module and UHC matching module. Standard AC automaton is used to scan content byte-by-byte as normal. And the other two modules are applied to accelerate scanning UHC, which are skipped if encountered. The mapping module is a bridge between AC automaton and UHC matching module. A set of up to 255 bytes from ASCII code is used for mapping identifiers. After a character is scanned, automaton will first search the character from the mapping set. If matched, the matched character will point to the corresponding UHC matching module. Afterwards, UHC matching module returns the state to AC automaton after the jump. As to UHC matching module, the literature adopts two UHC matching functions, hash table and fingerprint. Both can achieve a quick search. In different traffic environments, the model will choose the best one from the two functions. Note that UHC matching subsets and mapping set are one-to-one correspondence.

In terms of applicability, our model is versatile and flexible, and can be adjusted according to various scenarios. When the pattern set of UHC is scalable, UHC matching algorithms can be modified or replaced. In the scene of the normal pattern set with long length, the efficiency of AC is not optimal. In this case, an optimization algorithm such as path compression for AC can be used as an alternative. That said, as long as the model is finetuned, it can satisfy the specific requirements of different content providers and censorships.

2 Related works

Deep Packet Inspection (DPI) is a vital component of contemporary security and traffic analysis systems, and pattern matching algorithm is the heart of DPI. The essence of DPI performance is the efficiency of pattern matching algorithms. Conceptually, pattern matching algorithm contains two types: exact pattern matching and regular expression matching. On account of consuming a lot of resources for the latter with fewer practical applications, the literature focuses on the former.

The classic exact pattern matching algorithms are AC, WM and SBOM, which are commonly used in signature-based instruction detection [Snort (2019)]. The AC algorithm constructs a Deterministic Finite Automaton (DFA), that records the pattern set as a Trie tree. The AC automata is defined as follows:

$$M=(Q, L, g, f, qo, F) \qquad (1)$$

In Eq. (1), Q is a finite state set (denote all nodes in the trie tree), L is a limited list of input characters (denote characters on all edges of the trie tree), g is GOTO table, f is fail table, $qo \in Q$ is initiate state (denote root node) and F is the final state set (denote output table).

We shall give a brief overview of the original AC algorithm. The original AC algorithm includes three tables, the GOTO table, the FAIL table and the OUTPUT table. The GOTO table records the next state according to the current state and the next character. The FAIL table determines which state to return to when the next state obtained by the GOTO table is invalid. While the OUTPUT table saves the matched pattern in a state.

The original AC algorithm performs well in short pattern set. In the AC automata, if there are m nodes, the graph contains (*m-1*) GOTO transition edges and (*m-1*) failure pointers. To process an input string with *N* characters, the original AC algorithm will make at most *2N* state transitions. The main factor affecting the processing speed is the storage structure of the state transition graph. If the GOTO table of a node are stored in a linked list, then the original AC algorithm needs to carry out a sequential search to get the next state, which leads to more memory accesses and lower efficiency. Therefore, in the case of sufficient memory, array storage is an ideal choice. The performance can be improved by expanding the graph. A 2-D transition rule table is designed to store the next state, the elements of the table is a pair of <current state, input character>. Since the next state is predetermined and AC automata only consumes once memory access to search the next state.

The disadvantage of AC algorithm is that it takes up a lot of memory when the length of pattern is long, and the size of character set is large. In view of this case, WM has obvious advantages. Drawing on two hash tables, WM can quickly skip bad characters. While SBOM combines the advantages of both, it adopts a factor oracle structure to achieve search acceleration. Meanwhile, attacks against these algorithms are growing, Afek et al. [Afek, Bremler-Barr, Harchol et al. (2016)] proposed a threshold method to detect algorithm complex attacks and addressed it with multi-core architecture. Subsequently, Liu et al. [Liu, Shi, Zhang et al. (2018)] further improved the attack detection and demonstrated a two-step threshold detection method.

For the sake of high-speed DPI engine, researchers devoted themselves to explore innovative algorithms in both hardware and software implementations. The greatest throughput is hardware implementation [Meiners, Patel, Norige et al. (2010); Pao, Lin and Liu (2010); Chen and Wang (2013)], which relies on dedicated hardware, such as FPGA and CAM/TCAM. Nevertheless, it is expensive and not easy to update the program, so it is only suitable for ISP or large enterprises. On the contrary, software implementation Liu et al. [Liu, Zhang, Yu et al. (2018); Kumar, Dharmapurikar, Fang et al. (2006); Bremler-Barr, David, Harchol et al. (2012)] is universal. Additionally, with respect to the aforementioned work, our strategy can be applied on top of these work.

The aforementioned work is focused on the matching of desired content. As for UHC, Bremler-Barr et al. [Bremler-Barr, David, Harchol et al. (2015)] showed how repetitions in network traffic could be used to enhance DPI performance. A mechanism was proposed that changed the legacy AC algorithm, adding a dictionary of repeating data. The mechanism consisted of a slow path and a data path. The former recognized repetitions and created dictionary, while the latter traversed AC nodes at each step and determined how many characters to skip based on the dictionary. The solution achieved a throughput gain of 1.25-2.5 times the original throughput.

A close look at Barr's mechanism, although DPI performance had been improved, a significant gap remains in traversing each AC node. The dictionary is searched once for each node visited, which obviously increases the overhead of time. By comparing the size of character set between normal pattern set and UHC set, we find that the character set of UHC is smaller. In other words, it is not necessary to search dictionary for all nodes. Therefore, we design a small mapping set to construct a highway with AC node and dictionary. The mapping set is a character set. All the first characters of UHC set are extracted, and then a character set is composed after eliminating duplicate ones. Consequently, only partial AC node will execute the search of dictionary. On top of that, UHC strings are cut into k-grams in Barr's mechanism, we suppose the length of UHC string is L, there are (L-k -1) *k-grams* in all. The negative impact is the rise in hash collision rate due to the increase of new UHC sets. To address the problem, we improved the cutting method. Since the goal is to match UHC, that is, all characters can be hit, so the *k-grams* with overlapping is not necessary and it is sufficient to remain $\left\lfloor \frac{L}{k} \right\rfloor$ *k-grams*. In this way, the UHC set is greatly reduced and hash collision rate is also reduced.

3 Fast AC model with skipping

3.1 Model framework

Fast AC Model with Skipping (FAMS) is an enhanced AC algorithm model. There are three modules in FAMS, a standard AC automaton, a mapping module and UHC matching module. The model framework is shown in Fig. 1.

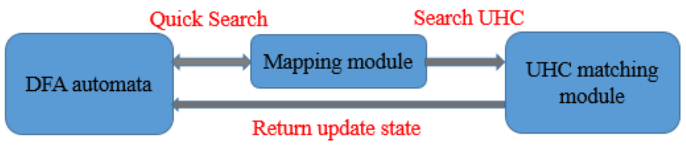

Figure 1: Model Framework

The mapping module is a quick search module, which is the key to improve DPI performance. In the module, a small mapping set is generated to connect AC nodes and UHC matching subsets. And the generation method is to select the first character of all string in UHC set, if a repetition exists, remains only one. When an AC node is accessed, quick search is performed to determine whether or not a secondary search of UHC set is required.

The UHC matching module mainly processes UHC matching and saves update state. The UHC matching algorithm uses hash function or fingerprint function. This module creates multiple subsets corresponding to the mapping set, and all the subsets use the same UHC matching algorithm. The saved update state is the next state returned to AC automaton. Each UHC string has a saved update state. If the suffix of UHC string is the same as the prefix of AC automaton, the state in the deepest level in AC automaton is to be saved. The state searching is as follows: a UHC string $P = \{p_1, p_2, \ldots, p_n\}$, its inverted string $Q = \{q_i | q_i = p_{n-i}, 0 \leq i \leq n\}$ is used as input. The AC automaton is traversed starting from root. When it is failed to match qi, this indicates that the currently traversed path is a longest suffix of an UHC string. And then the current state is recorded as update state of string *P*.

The process of FAMS: inspected traffic traverses the trie of AC automaton. For the current node, searching the current character in mapping set, if not found, continue traversing next character, else further searching UHC set and return update state to AC automaton if success.

3.2 UHC matching algorithm

In Barr's mechanism, the patterns of UHC set are *k-grams* with the same length. The patterns are stored as an open hash table and a Bloom filter is used to query. Nevertheless, the Bloom filter uses only a single hash function that degrades to a normal hash. In addition, the *k-grams* are all substrings of UHC, that is, if the length of a UHC string is *L*, then there is (*L-k-1*) *k-grams*. So many *k-grams* will undoubtedly increase the probability of hash collision. Owing to the mapping set introduced by FAMS, UHC set is divided into subsets by the first character, which can reduce conflicts. Further, we chop the number of *k-grams* to $\left\lfloor \frac{L}{k} \right\rfloor$ by deleting *k-grams* with partial overlap. This operation greatly reduces the size of UHC set and hash collision, it is very beneficial to improve performance. When chopping *L* to *k-grams*, *L* may be divided by k with remainder, the remainder is merged into the last k-gram. As shown in Fig. 2, the UHC string "*GoogleScholar*" is divided by *4-gram* and the subset "*fGoog*", "*leSh*", "*holarg*" is obtained.

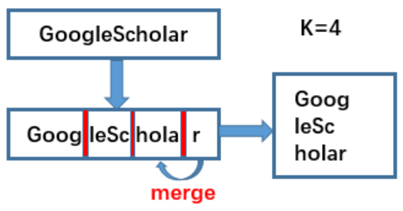

Figure 2: Generate *k-gram* subsets

More specifically, given UHC string *L* with length *l* and some constant *k*, we wish to divide *L* into *n* substrings to get UHC subsets $\bar{L}$.

$$n = \left\lfloor \frac{l}{k} \right\rfloor, 0 \le i < n-1 \quad (2)$$

$$L = \{k_i | L_{i\times k+1}, L_{i\times k+2}, \dots, L_{i\times k+k}\} \cup L_{mod} \quad (3)$$

$$L_{mod} = \{L_{(n-1)\times k+1}, \dots, L_l\} \quad (4)$$

$$Let\ k_{n-1} = k_{n-1} \cup L_{mod} \quad (5)$$

$$\bar{L} = \{k_0, k_1, \dots, k_{n-1}\} \quad (6)$$

The literature adopts two UHC matching algorithms: hash table and fingerprint. The common feature of both is that the window size is the shortest length of UHC set. The substring in the window represents the entire pattern. Furthermore, the substring set with the minimum collision rate is used as the representative set. The selection algorithm of substrings is as described in Algorithm 1.

Algorithm 1: Selection of Substring

```
INPUT   UHC set $K = \{k_1, k_2, \dots, k_m\}$ with length $k$ , $P = \{p_1, p_2, \dots, p_n\}$ with length $L = \{l_1, l_2, \dots, l_n\}, k < l_i < 2 \times k$
OUTPUT  New UHC set $\bar{C} = \{\bar{c_1}, \bar{c_2}, \dots, \overline{c_{m+n}}\}$, hash table H $= \{h_1, h_2, \dots, h_{m+n}\}$
1.  $\bar{C} \leftarrow \emptyset, H \leftarrow \emptyset, i \leftarrow 1, j \leftarrow 1$;
2.  \\ compute unique ID($h_i$) for set K
3.  while $i \le m$ do
4.      $h_i \leftarrow Func(k_i)$ ;
5.      if $h_i \in H$ then
6.          $\bar{C} = \bar{C} \cup k_i, H \leftarrow H \cup h_i$;
7.      else
8.          $\bar{C} \leftarrow \emptyset, H \leftarrow \emptyset, i \leftarrow 1$;
9.          Update function or hash space;
10.     end if
11. end while
12. \\ compute unique ID($h_j$) for set P
13. $P_{jz} = \{p_{jz}, p_{jz+1}, \dots, p_{jz+k+1}\}$;
14. while $j \le n$ do
15.     while z $\le l_j - k + 1$ do
16.         $h_j \leftarrow Func(P_{jz})$;
17.         if $h_j \in H$ then
18.             $\bar{C} \leftarrow \bar{C} \cup P_{jz}, H \leftarrow H \cup h_j$;
19.             break;
20.         end if
21.         $z \leftarrow z + 1$;
22.     end while
23. end while
```

For example, in Fig. 3, suppose that DKJMEF and D6DFEK are in conflict, D6DFEK will be replaced by another random substring with different first character. The selection of substring requires several rounds ($1 \le r \le k$) of computation. Accordingly, the process takes a lot of time. However, it is less significant, as it belongs to the preliminary work before NIDS runs.

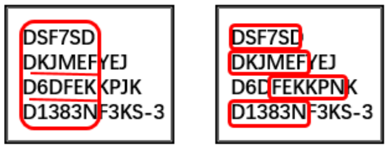

Figure 3: The selection of substring

On the flip side, there are $P_r = P_L - k$ characters left after substring selection, which have to be processed by AC automaton. $0 \leq P_r < k$, it has little impact on the overall performance, so it is tolerable.

With regards to matching function, hash table and fingerprint. Hash table is an open hash table, it only needs to select the corresponding hash number according to length k. Fingerprint is a polynomial function, the definition is emphasized below.

Definition 1: $\varphi(p)$ is the fingerprint of string p, if and only if $\varphi(p)$ satisfies two conditions:

(1) φ is the function of string p or its substring, if two strings or substrings of their fingerprints are identical, and then their fingerprints are identical.

(2) For any two strings $p_1 \neq p_2, \rho_r\left(\varphi_{f,\sigma}(p_1) = \varphi_{f,\sigma}(p_2)\right) \ll 1$, that said, the probability that p1 and p2 have the same fingerprint is much less than 1.

Definition 2: for the given string $p = p_1, p_2, \dots, p_m, \sigma \in \delta(N^4)$, a polynomial fingerprint function for p is $\varphi_{f,\sigma}(p) = (\sum_{i=1}^{m} p_i f_i \; mod \; \sigma), 1 \leq i \leq m, f_i \in F_\sigma$, where σ is primer number, $F_\sigma = \{\forall f_i \in F_\sigma | f_i = mod(\mathbb{Z}, \sigma)\}$.

A polynomial fingerprint has two properties:

(1) Fingerprint $\varphi(p_1, \dots, p_m, p_{m+1})$ can be calculated according to $p_1, p_2, \dots, p_m, \varphi(p_1, p_2, \dots, p_m)$ and p_{m+1}.

(2) When $1 \leq i \leq m, \varphi(p_{1+i}, p_{2+i}, \dots, p_m)$ can be calculated according to $p_1, p_2, \dots, p_m, \varphi(p_1, p_2, \dots, p_m)$ and $\varphi(p_1, p_2, \dots, p_i)$.

In the actual operation of FAMS, the matching algorithm can be selected according to the conflict rate test result of different UHC sets.

3.3 Working instance

In order to clarity how FAMS works, we elaborate on the details through an instance. Assume that the normal pattern set is {*E, BE, BD, BCD, JDBC*}, the corresponding AC automaton is shown in Fig. 4 on the left. What's more, we assume UHC set is {*BTAGXUBC, BCDZLA30TKN, JAVASCRIPT, JASONTK101B*}, which is divided into two separate subsets by the first character, {*BTAGXUBC, BCDZLA30TKN*} and {*JAVASCRIPT, JASONTK101B*}. The window sizes of the two subsets are 8 and 10, respectively. After the calculation by hash or fingerprint, we get two pattern subsets as UHC sets. In each UHC set, all patterns have the same length. The generation process of UHC sets is depicted in Fig. 5. Afterwards, we figure out mapping set {*B, J*}.

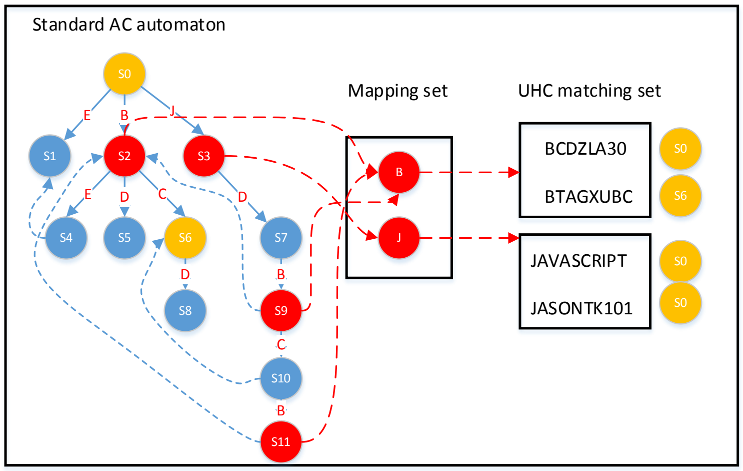

Figure 4: FAMS working diagram

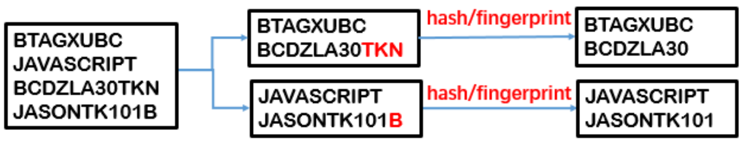

Figure 5: UHC subsets generation process

As to update state in UHC matching records, a prefix of normal pattern is a suffix of an UHC string. In Fig. 5, the suffix BC of the second UHC string is a prefix of BCD in normal patterns.

After finishing UHC matching set and mapping set, we move the perspective to standard AC automaton. The automaton scans from the initial state s_0, when an input character is in mapping set, then searching the corresponding UHC subset according to mapping relation. If not found, automaton keeps scanning, else UHC matching module returns update state to the automaton. This moment, a judgment is needed to determine whether to jump state.

Judgement condition: Assume current character is pi that is in mapping set, the window size of the corresponding UHC is k. The scanning at automaton continues to search the bytes p_i, p_{i+1}, p_{i+k-1} one by one, it is paused until reaching such a state with an input byte p_{i+j} , whose depth is less than or equal to j.

If the condition is satisfied, the current state jumps to the update state saved in UHC string p_i, p_{i+1}, p_{i+k-1}.

Everything is ready now, we induce the input *JDBCBTAGXUBCDH*. The first character is matched mapping set, but not matched UHC subset. The scan continues until state s_{10}. Then, *BTAGXUBC* is in UHC subset. Owing to the next character is *B*, and new current state is s_{11}, whose depth is 5>1 (index of UHC is 1). Since the judgement condition is unsatisfied, the next character *T* is entered. By now, current state is s_0, whose depth is 0<2 (index of UHC is 2), condition establishment. Therefore, the update state s_6 in UHC subset is new current state. FAMS skips the rest six characters *AGXUBC* and continues to scan *DH*. Finally, the pattern *BCD* is matched successfully. The scanning process is presented in Tab. 1 and the characters with underline are skipped.

Table 1: Scanning process for input *JDBCBTAGXUBCDH*

p_i	J	D	B	C	B	T	<u>A</u>	<u>G</u>	<u>X</u>	<u>U</u>	<u>B</u>	<u>C</u>	D	H
match	✗	✗	✗	✗	✓	-	-	-	-	-	-	-	✗	✗
state after p_i	s_3	s_7	s_9	s_{10}	s_{1}	s_0	-	-	-	-	-	s_6	s_8	s_0
depth	1	2	3	4	5	0	-	-	-	-	-	2	3	0
index of UHC	-	-	-	-	1	2	-	-	-	-	-	-	-	-

Next, Algorithm 2 gives the pseudo-code for FAMS scanning process.

```
Algorithm 2: FAMS SCAN
INPUT    $P = (p_0, p_1, \dots, p_{n-1})$
OUTPUT   current state $s_{cur}$
1.   $s_{cur} \leftarrow s_0, i \leftarrow 0$;
2.   while $i < n$ do
3.       if $p_i \in S_{mapping}$ then
4.           $w = Win_{S_{UHC}}, j \leftarrow 0$
5.           $f = Func(p_i, p_{i+1}, \dots, p_{i+w-1})$
6.           if $f \in S_{UHC}$ then
7.               while $s_{cur}.depth > j$ and $i < n$ do
8.                   $s_{cur} \leftarrow scan(s_{cur}, p_i)$;
9.                   $i \leftarrow i + 1, j \leftarrow j + 1$;
10.              End while
11.              $i \leftarrow i + (w - j)$;
12.              $s_{cur} \leftarrow S_{UHC}.s_{update}$;
13.          else
14.              $s_{cur} \leftarrow scan(s_{cur}, p_i)$;
15.              $i \leftarrow i + 1$;
16.          end if
17.      end if
18.  end while
19.
```

4 Experimental results

In our experiment environment, physical memory is 16 G, network card is a gigabit and CPU is Intel Core i7. To improve throughput, the platform is driven by zero-copy unlocked DPDK. Our model runs with 8 threads in parallel.

The number of the normal pattern is 10^5, which is selected from Snort rules and generated randomly according to Snort rules. While UHC set is extracted from gateway traffic of Harbin Institute of Technology. A script is used to extracted duplicate strings and ranks them. We removed the strings that conflicts with normal pattern set and chose the top 200 as UHC set. A close look at UHC set, it is obvious that HTTP header fields rank top, as shown in Tab. 2.

Table 2: The top of UHC set

Ranking	UHC	Ranking	UHC
1	text/html	6	text/xml
2	text/plain	7	application/xml
3	image/jpeg	8	gzip
4	image/png	9	Keep-alive
5	Image/jif	10	Mozilla

With regards to induce traffic, we captured five files with the sizes about 943 M, 1.3 G, 2.2 G, 2.8 G, 4.5 G and numbered the files numerically (1-5). To measure the performance of our proposed FAMS, we recorded the following information for analysis: throughput, automata node access, UHC node access and memory occupancy. The object of comparison is the original AC and Birr's double path. Accordingly, a total of three programs were deployed.

Throughput: The results of throughput comparison are shown in Fig. 6. The x-coordinate represents the file number and the y-coordinate represents the average throughput. Our experiments show promising results that we achieve a throughput gain of 1.3-2.6 times the original AC and 1.1-1.3 times Barr's double path method.

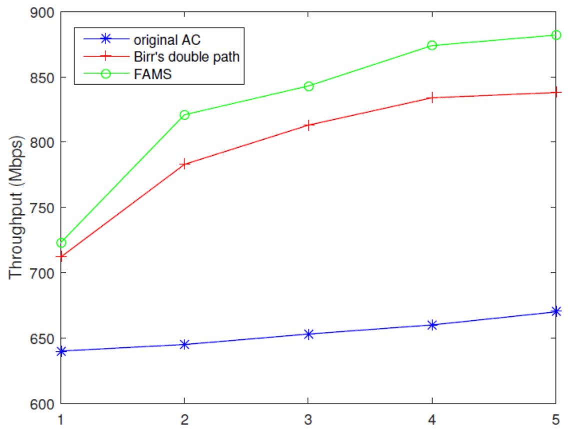

Figure 6: Throughput comparison

Memory occupancy: In the three programs above, the order of memory occupancy is FAMS $\approx$ Birr's>original AC. FAMS has 5 K more memory than the original AC, accounting for only 0.5% of the total memory of the automaton. Therefore, a small increase in memory has less impact on performance.

Automata node access: Automata is the bottleneck of system performance, and the frequency of its node access reflects memory access which directly affects the performance. The fewer visits, the better performance. The statistics of the number of visits to a node includes two parts: AC automaton node and UHC node. We run five test files in the experiment. From Tab. 3, we can see that no matter which file, the result is the same. The original AC is the most, Birr is the second, and FAMS is the least. Thus, a large number of repetitive strings in the input data are hit in FAMS, resulting in skipping many characters.

Table 3: The top of UHC set

File number	1	2	3	4	5
Original AC	65	892	191	247	389
Birr's	62	785	181	236	371
FAMS	61	83	178	231	365

***UHC node access*:** Statistical results of UHC nodes are shown in Tab. 4. UHC node access accounts for 5%-7% of all nodes. The total number of bytes skipped accounts for 7%-8% of the data. Undoubtedly, it is benefit for performance improvement.

Table 4: Statistics of UHC node access (unit: $\mathbf{10^7}$)

File number	1	2	3	4	5
UHC statistics	3.5	4.9	13.8	15.7	23.3

5 Conclusion

The current boost traffic poses a challenge to DPI performance, most researchers work on the optimization of the DPI algorithm itself. However, according to the observation of traffic content, there is a lot of undesired high-frequency content, which is no needed to match. In this literature, a fast AC model with skipping (FAMS) is proposed to accelerate AC automaton by skipping repetitions. Based on the method of Birr's double path, FAMS adds mapping set so that UHC set is searched only if the input characters in mapping set. Another, for a UHC set, UHC algorithm chooses an appropriate one from hash and fingerprint functions. Unlike simply traversing AC automaton, we try to make sure early whether skipping is possible, if it is, no scanning some characters. Finally, our experiments show that FAMS is beneficial to the improvement of DPI performance.

Acknowledgement: This work was supported by National Natural Science Foundation of China under Grant (Nos. 61771166, 61771166, 61402137) and National Key Research & Development Plan of China under Grant 2016QY05X1000.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Afek, Y.; Bremler-Barr, A.; Harchol, Y.; Hay, D.; Koral, Y. (2016): Making DPI engines resilient to algorithmic complexity attacks. *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3262-3275.

Aho, A. V.; Corasick, M. J. (1975): Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, vol. 18, no. 6, pp. 333-340.

Allauzen, C.; Crochemore, M.; Raffinot, M. (1999): The Factor oracle: a new structure for pattern matching. *International Conference on Current Trends in Theory and Practice of Computer Sci*ence, pp. 295-310.

Bremler-Barr, A.; David, S. T.; Hay, D.; Koral, Y. (2012): Decompressionfree inspection: Dpi for shared dictionary compression over HTTP. *IEEE International Conference on Computer Communications*, vol. 131, no. 5, pp. 1987-1995.

Bremler-Barr, A.; David, S. T.; Koral, Y.; Hay, D. (2015): Leveraging traffic repetitions for high-speed deep packet inspection. *IEEE International Conference on Computer Communications*.

Chen, C. C.; Wang, S. D. (2015): A hybrid multiple-character transition finite-automaton for string matching engine. *Microprocessors & Microsystems*, vol. 39, no. 2, pp. 122-134.

Chen, C. C.; Wang, S. D. (2013): An efficient multicharacter transition string-matching engine based on the aho-corasick algorithm. *ACM Transactions on Architecture & Code Optimization*, vol. 10, no. 4, pp. 1-22.

Kumar, S.; Dharmapurikar, S.; Fang, Y.; Crowley, P.; Turner, J. (2006): An algorithms to accelerate multiple regular expressions matching for deep packet inspection. *Acm Sigcomm Computer Communication Review*, vol. 36, no. 4, pp. 339-350.

Liu, P.; Liu, Y. B.; Tan, J. L. (2005): A partition-based efficient algorithm for large scale multiple-strings matching. *Computers, International Symposium on String Processing and Information Retrieval*, pp. 399-404.

Liu, L. K.; Zhang, H. L.; Yu, X. Z.; Xin, Y.; Shafiq, M. et al. (2018): An efficient security system for mobile data monitoring. *Wireless Communications and Mobile Computing*, vol. 2018, pp. 1-10.

Liu, L. K.; Shi, J. T.; Zhang, H. L.; Yu, X. Z. (2018): Tearing down the face of algorithmic complexity attacks for DPI engines. *IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications*, pp. 751-758.

Meiners, C. R.; Patel, J.; Norige, E.; Torng, E.; Liu, A. X. (2010): Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. *Proceedings of the 19th USENIX Conference on Security*.

Nemls, T.; Ahamad, M. (2010): The packet scheduling for deep packet inspection on multi-core architectures. *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications System*, pp. 21-30.

Pao, D.; Lin, W.; Liu, B. (2010): A memory-efficient pipelined implementation of the aho-corasick string-matching algorithm. *IEEE ACM Transactions on Architecture and Code Optimization*, vol. 7, no. 2, pp. 10.

Snort (2019): Snort intrusion detection system. http://www.snort.org.

Tan, G. M.; Liu, P.; Bu, D. B.; Liu, Y. B. (2011): A Revisiting multiple pattern matching algorithms for multi-core architecture. *Journal of Computer Science and Technology*, vol. 26, no. 5, pp. 866-874.

Wu, X. N.; Zhang, C. Y.; Zhang, R. L.; Wang, Y. J.; Cui, J. H. (2018): A distributed intrusion detection model via nondestructive partitioning and balanced allocation for big data. *Computers, Materials & Continua*, vol. 56, no. 1, pp. 61-72.

Wu, S.; Manber, U. (1994): A fast algorithm for multi-pattern searching. http://webglimpse.net/pubs/TR94-17.pdf.

Xing, W.; Pao, D. (2018): Memory-based architecture for multi-character aho-corasick string matching. *IEEE Transactions on Very Large-Scale Integration Systems*, vol. 24, no. 99, pp. 1-12.

Yuan, X.; Duan, H.; Cong, W. (2018): Assuring string pattern matching in outsourced middleboxes. *IEEE Transactions on Networking*, vol. 26, no. 3, pp. 1362-1375.