ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **23**, Number T, 2020, T94-T105

HARD: Bit-Split String Matching Using a Heuristic Algorithm to Reduce Memory Demand

Xun LI[1, 2], Lishui CHEN[2] and Yazhe TANG[1]
[1] Xi'an Jiaotong University, No.28, Xianning West Road, Xi'an, 710049, P.R. China
[2] Science and Technology on Communication Networks Laboratory, Shijiazhuang , 05000, P.R. China
Email: lixun2011@stu.xjtu.edu.cn, 78015159@qq.com, yztang@mail.xjtu.edu.cn

Abstract. High-speed content inspection relies on a fast multi-pattern matching algorithm to detect predefined rules. When the number of target rules becomes large, the memory requirements of the matching engine become a critical issue. An effective technique to design high-performance matching engines is to divide the target rule set into multiple subgroups and to use a parallel matching hardware unit for each subgroup. The key to this effective technique is how to find a strategy to divide subgroups. This paper proposes an effective rule classifying method referred to as HARD for heterogeneous bit-split string matching architectures. HARD uses the uniqueness of the target pattern to classify all target rule characters. This paper also presents a method to estimate the distance between strings in unique pattern category. The distance formula is next used to find a class for each rule. Furthermore, each class will be processed on different sizes of finite state machine. The experimental results show that the more the number of rules in the rule set, the more obvious the effect of HARD. In popular data sets, when the number of rules is above 4000, HARD can save nearly 50% of memory consumption compared to the previous bit-split string matching methods mentioned in the paper.

Keywords: String Matching; State Machine Splitting; Finite State Machines; Fine-grained Traffic Identification

1. Introduction

The emerging cloud computing and mobile internet are quite successful in business, with thousands of new applications generating huge network traffic. The traditional traffic identification reports only at the protocol or application granularity, e.g., the traffic belongs to Weibo or QQ, but it fails to provide fine-grained traffic identification to tell which specific function (e.g., login, data input, output, etc.) the user is operating. Therefore, fine-grained traffic identification is essential for service providers and network operators to measure and monitor the network at a deeper level. This is an important part of ensuring network security and quality of service (QoS).

Nowadays, one of the most powerful methods of ensuring network QoS is Deep Packet Inspection (DPI), which analyzes the payload to determine whether the target rules in the application layer match. In many cases, the target rule consists of multiple strings; therefore, the string matching engine is a key part in the DPI process. With the development of network services, the number of target rules has increased, and the length of the rule string (which means the number of characters in it) may vary greatly. After that, length change will lead to increase space usage. This poses a new challenge to the network matching engine. The most important question is: How to design a space-saving matching engine under fine-grained network recognition conditions?

On this issue, previous researchers have done a lot of effective research work. A string matching method based on deterministic finite automaton (DFA) is proposed. In addition, in order to speed up the processing, the researchers proposed hardware structures such as FPGA [1]–[3] and TCAM [4] and made a lot of optimizations to reduce memory requirements. Despite it has many advantages, sparse memory usage is one of main problems in DFA-based string matching. Finite state machine (FSM) based algorithms such as Aho-Corasick (AC) are widely used in string matching hardware design. These memory-based algorithms can handle problems in a linear time complexity in time. However, memory requirements that are proportional to the number of states and the number of bits in each state can be an important limiting factor [6]. Khan et al. [6] and Jony [7] presented comparative analysis and experimental results on various string matching algorithms. In the AC algorithm described in [5], the number of state pointers in each state is 256 for ASCII (8-bit) input, which results in DFA-based string matching engines requiring a lot of memory.

The following scenarios have motivated us in this paper to try to solve the problem mentioned above.

- Bit-split string matching [8]. Use multiple FSM TILEs with multiple input bit groups to reduce the number of state pointers. Tan L et al. [8] presents a bit-split string matching engine. The output state contains a partial match vector (PMV); the full matching vector (FMV) is obtained by performing a bitwise AND operation between the PMVs from the FSM tiles. Each bit in the FMV indicates whether its own pattern matches.
- Pattern grouping [9]. Another effective way to improve the efficiency of FSM-based string matching is to divide the target string into multiple subgroups. This will lead to replacing large chunks with multiple TILE working in parallel. Each string subgroup is mapped to a TILE. In general, assigning strings to subgroups can significantly improve performance in matching and overall memory consumption.

The existing pattern grouping algorithms are aimed at a homogeneous architecture composed of the same TILE. A new method called a Heuristic Algorithm to Reduce memory Demand (HARD), is proposed in this paper. We first use the uniqueness of the target pattern [9] to classify all target rule characters into two categories and target a heterogeneous architecture, where each TILE can have a different number of memory resources and configurations. This makes resource allocation more efficient, which significantly reduces overall memory usage.

In general, two aspects of this manuscript contribution can be listed as follows. First, we use the new heterogeneous finite state machine architecture. Second, a grouping heuristic algorithm for heterogeneous bit-split string matching architecture is proposed. This can allocate resources more efficiently, further reducing memory usage. The remainder of the paper is organized as follows. Section 2 describes the related work in the string matching area. Section 3 introduces the overview of HARD and describes the details of the HARD. We give the experimental results in Section 4. Finally, Section 5 summarizes the paper and discusses the future work.

2. Related work

String matching schemes classified into three types: heuristic-based, filtering-based, and automaton-based string matching schemes [10].

Heuristic-based: The main idea of a heuristic-based string matching scheme is to skip unmatched characters and accelerate the search. A famous heuristic-based string matching schemes is the Boyer-Moore algorithm [11]. Searching for the occurrence of a pattern by performing comparisons at different alignment positions of the pattern and the text to be searched. With the information obtained by the preprocessing mode, many sequences can be skipped. In the worst case, however, there are no skipped characters. Assuming the pattern length and number of characters in the text are m and n, the worst case time complexity is *O(nm)*. In addition, the BM algorithm is not suitable for multiple patterns matching. Although multiple pattern-matched strings can be used in heuristic-based string matching [13], multiple processing elements or cores should be configured. The use of heuristic algorithms to solve problems in these papers [12]–[16] is also worth learning. Abed Alguni [15] introduces two improvements to improve Cuckoo search and provides a good idea in terms of heuristic algorithms. Over the past decades, optimization problems have grown ever more popular not only in the academic literature but also in practice. Mls [14] and Precup et al. [15] provide some new idea in terms of optimization algorithms.

Filtering-based: Since parallel strings match multiple patterns, a filtering-based string matching scheme is preferred. Filter-based string matching schemes use hash [17] or bloom filter [18]. These two algorithms have high memory efficiency when processing bit vectors. Filter-based string matching schemes can quickly exclude input data that does not contain a pattern to match. The assumption of this scheme is that there are few matching patterns in the text to be matched. In the worst case, when the matching is frequent, there are not many characters that can be skipped, and the performance of the algorithm given to the filtering will not be exerted. In the worst case, filtering-based string matching schemes might suffer from algorithmic attacks.

Automaton-based: In an automatic string matching scheme, multiple patterns are mapped using states and state transitions between states. In particular, DFA-based string matching schemes perform a fixed number of state transitions at a time. Therefore, linear worst-case performance can be guaranteed. In addition, target patterns can be matched across multiple payloads, as each state contains information from the input sequence. However, DFA-based string matching schemes require a large amount of memory to store state information and state transitions for each state. The paper [19] proposed a dictionary ordinal string search algorithm based on compressed DFAs. In the AC algorithm, the DFA should contain a failure pointer from each state to the longest suffix state or matching sub-pattern. By sharing a common prefix, the number of states in DFA can be reduced; therefore, the state information can be compressed. However, in the traditional AC algorithm, since the number of state transitions is large, the memory requirement for storing state transitions in each state is large.

Automaton-based string matching scheme can be implemented using general memory. For example, the AC algorithm is implemented using a two-dimensional memory architecture. Since storage state transitions require a large amount of memory, some studies have been performed in order to reduce the number of storage state transitions. In [1], [20], [21], by using configurable logic and distributed memory in FPGA, only state transitions to non-initial states are stored. However, compared to memory-based string matching engines, FPGAs are not flexible and updatable. In [5], [22], TCAM is used to compress the state transition information in DFA-based string matching. However, due to the high price and power consumption of TCAM, the application of TCAM in DFA-based string matching is limited.

In [8], a bit-split DFA-based string matching was proposed to reduce the memory requirement for storage state transitions. To reduce the total number of state transitions for each state, multiple DFAs can be constructed for each input bit group by splitting one or more ASCII character codes into several bit groups. In addition, in order to identify matching target patterns, each state should contain its own matching vector and a set of bits or PMVs, where the value of each bit indicates whether the relevant target pattern matches in the state. Therefore, the memory requirements for storing match vectors can be very large.

In general, grouping strings can significantly improve the performance of matching and overall memory consumption. Pattern grouping based on lexicographical sorting [8] aims to reduce the total number of states required in FSMs by maximizing the shared common prefixes among the strings of each group. To do this, these methods sort strings based on prefixes and then assign rule strings to different groups in order. Kim et al. [9] introduced a pattern 9grouping algorithm that uses the average length of the strings as grouping standard. The goal of Kim's algorithm is to balance the number of target strings mapped onto each SMU. For this purpose, it makes the average length of the strings in each subgroup as close as possible to the overall average length of the strings.

In the previous works [8], [9], [23], [24] and [25], several architectures based on the bit-split string matching scheme were proposed. In [12], [26], the memory requirements for storing state transitions towards initial states are reduced. In [8], [26], a multi-byte string matching is performed by multiplying the number of memory blocks. Even though the previous works related to the pattern grouping and architectures mentioned above have developed new bit-split string matching schemes, a string matcher should have memory blocks for storing match vectors, which can be the disadvantage in the new architectures based on the bit-split string matching scheme.

3. Proposed method HARD

In order to solve the problem presented in Section 1, we designed an effective method, called HARD that to divide the target rule set into multiple subgroups and to use a parallel matching hardware unit for each subgroup. As shown in Fig. 1, the input to HARD is rules set. In the proposed scheme step1, all target patterns are categorized into two groups - a set with non-unique patterns and a set with unique pattern. After that, in the step2, there is two phases: 1) estimation phase, which uses a calculation to estimation the correlation between rules and 2) growing phase for mapping rules to subgroups. The following subsections describe the method in detail.

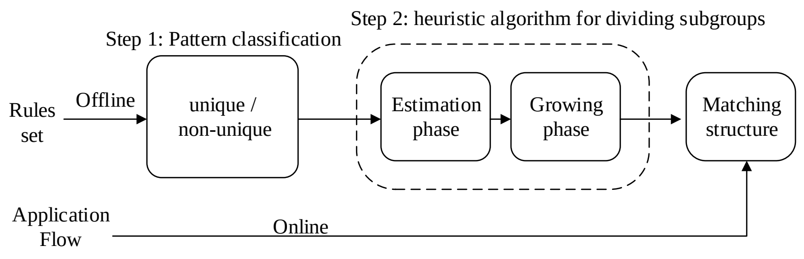

Fig. 1. Proposed method HARD overview

3.1. Changed target architecture

In order to reduce the memory requirement for storage state transition, a string matching method based on bit-split DFA was proposed in [8]. By splitting one or more ASCII character code into several bit groups, multiple DFAs can be constructed for each input bit group, thereby reducing the total number of state transitions for each state. The paper [8] also describes a high-throughput architecture string matching engine. Multiple homogeneous string matchers are executed in parallel, mapping a small number of target patterns to each string matcher. In order to identify matched target patterns in states, each state should contain its own matching vector and a set of bits or PMVs, where the value of each bit indicates whether the related target patterns are matched or not in the state. In this method, the memory requirements for storing match vectors might be very large. FMV memory usage is still one of main problems in this architecture.

To overcome the problem of memory usage waste, this paper tries to optimize memory in two ways. Utilizing the uniqueness of pattern technique proposed in the paper [26], we divide the rule set into two categories, which can reduce the FMV space of some rule strings, thus saving memory space; In addition, we believe that the homogeneous FSM will have a waste of space, because the size of the FSM is determined by the rule with the largest number of FSM state nodes. So, we have proposed heterogeneous FSM. To reduce the memory usage by sharing as many state nodes as possible under the bit-split FSM. Although such architecture adds complexity to the hardware circuitry, we believe this approach is feasible on some memory-intensive machines. We present the new bit-split string matching architecture, Fig. 2 shows the schematic of the entire matching process.

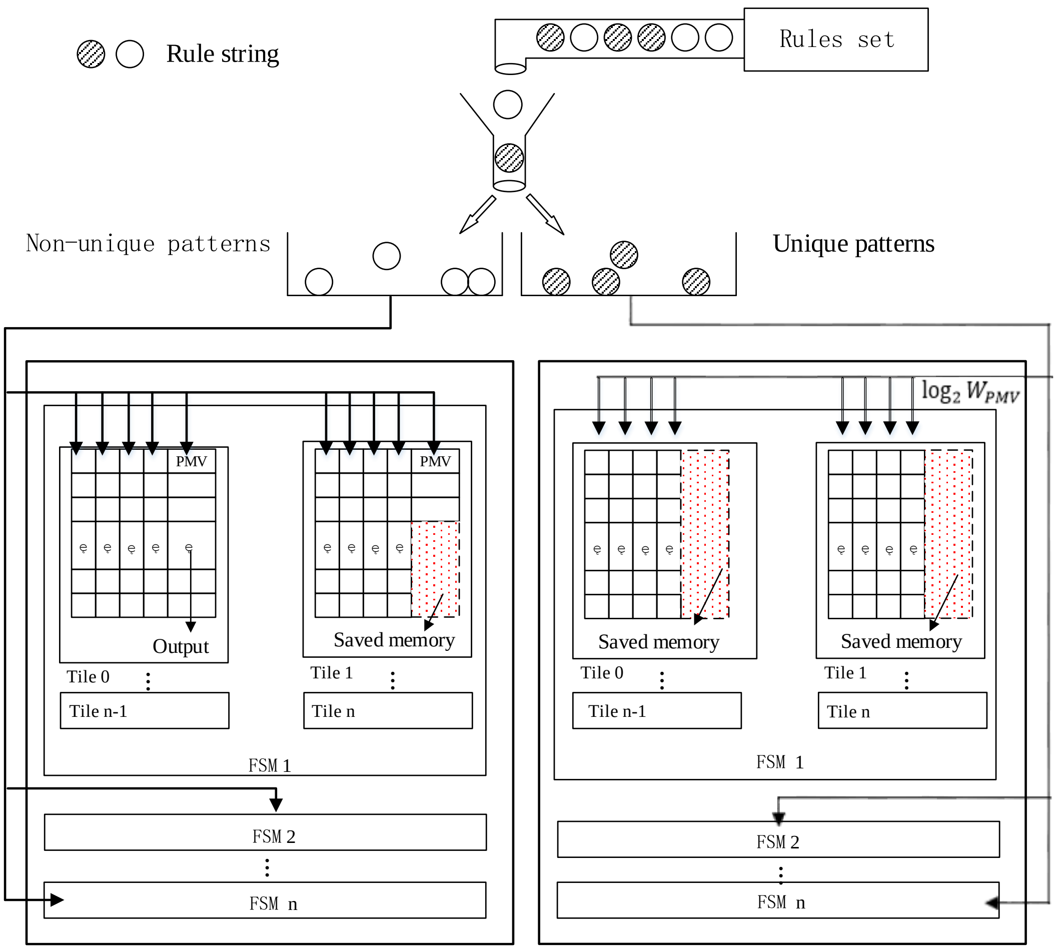

Fig. 2. Proposed architecture with heterogeneous FSM

Fig. 2 also shows the proposed architecture with heterogeneous FSM. The upper half part is a classifier, in which the strings of the rule set are divided into two groups according to the unique pattern concept for subsequent processing. The lower part is a complete device, comprised of a set of rule modules. Each rule module acts as a large state machine and is responsible for a group of rules. Each rule module is composed of a set of TILEs (n TILEs are shown in this figure). Each TILE is essentially a table with some number of entries and each row in the table is a state. Each state includes some number of next state pointers and a partial match vector of different length. Fig. 3 shows that a rule module can take some number of bytes as input at each cycle and output the logical AND operation result of the partial match vectors of each tile. This technique splits characters into multiple b-bit chunks, where b=1, 2, 4 or 8, and uses one FSM to process each chunk. When b=1, one FSM is utilized to process each single bit input character, and when b=8, the method is reduced to the original character-level AC algorithm.

We use the heterogeneous FSM architecture. The different FSMs in each tile apply for the corresponding PVM space according to the number of different FSM states. The structure designed in this way is to save memory usage space. In terms of unique pattern, we do not need PVM space consumption. Because for a unique set of patterns, each state matches only one pattern. The bit-splitting string matcher only needs to store the pattern matching index, instead of storing PMVs, which can reduce memory requirements. So, we only need the same number of bits as the FSM state.

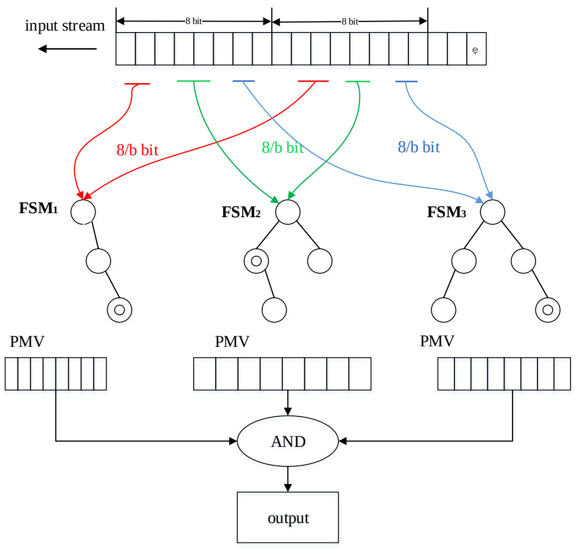

Fig. 3. In the matching process, each state machine matches a fixed position, and multiple state machines work in parallel at the same time.

3.2. Core idea and optimization

The memory consumption can be estimated from the architecture as shown in Fig. 1. The minimum memory requirements for the nth FSM of the m^{th} TILE is as follows:

$$M_{FSM_{mn}} = \left(2^b \left\lceil \log_2 S_{num_n} \right\rceil + W_{PMV_m}\right) S_{num_n} \qquad (1)$$

where b is the number of bits processed at a time in an FSM. S_{num} and W_{PMV} respectively represent the total number of states and PMV bit-width. The proposed approach uses formula (1) to calculate the minimum memory requirement for each FSM. The bit-width of the PMV that is equal to the number of strings mapped onto that TILE. So, the grouping scheme can significantly impact the $stateNum_n$ factor. The proposed algorithm explores the search space to find the grouping scheme that minimizes total memory consumption. Total memory consumption can be list as

$$T_{Mem} = \sum_{m=1}^{G} \sum_{n=1}^{8/b} M_{FSM_{mn}} \qquad (2)$$

where G is the number of groups. To save memory, we can optimize by reducing both the value of W_{PMV} and S_{num}.

To reduce the value of W_{PMV}, we classify the rule strings in the original rule set using the unique and non-unique patterns defined in [9]. Kim H. et al. [9] propose a memory-efficient DFA-based string matching scheme that reduces the memory requirements by not storing the matching vectors.

To reduce the value of S_{num}, One question we encountered was: How to assign rules to different classes to reduce the number of nodes in the bit-split DFA? And finally achieve the minimum memory consumption. We define it as an Assign Rules to different Classes Problem (ARCP). Suppose that the field i has a new allocation sequence $\psi(i)$, $i = 1 \ldots N$, after the organization, and the memory consumption in field i is $D_{\psi(i)}$. So ARCP seeks $\psi(i)$, $i = 1 \ldots N$, to minimize the total cost

$$\min_{\psi \in T_N} \sum_{i=1}^{N} D_{i\psi(i)} \qquad (3)$$

where T_N is the set of all the permutation.

In fact, this problem in our paper can be reduced from Quadratic Assignment Problem (QAP) [26] in polynomial time, denoted as QAP $\leq_p$ ARCP. QAP considers allocating n facilities to n locations. There are three types of costs in the problem: the cost function of the distance between locations, the flow function between facilities, and the cost of placing the facility in one location. The goal of QAP is to minimize the total cost of allocating facilities to different locations. If we remain the first two costs in QAP and set the cost of a facility-location placement to be zero, the problem can be formulated as

$$\min_{\phi \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\phi(i)\phi(j)} \tag{4}$$

where $\phi(i), i = 1 \dots n$ is the location used to placing facility i, f_{ij} and $d_{\phi(i)\phi(j)}$ are the cost function related to flow between facilities and distance between locations.

To reduce QAP to ARCP, we first rewrite the ARCP formulation in (3) as

$$\min_{\psi \in T_N} \sum_{i=1}^{N} \sum_{j=1}^{N} D_{ij} I_{\psi(i)\psi(j)} \tag{5}$$

$$I_{\psi(i)\psi(j)} = \begin{cases} 1, & \text{if } j=\psi(i) \\ 0, & \text{otherwise} \end{cases}$$

If we map the original fields order (of ARCP) to facility (of QAP) and the organized assign order (of ARCP) to the location (of QAP), (4) can be transferred to (5) by mapping the variables and functions: $n \to N, f \to D, d \to I, S_n \to T_N, \phi \to \psi$. Since QAP is known as an NP-hard problem [276, ARCP is also NP-hard with a polynomial reduction from QAP.

The brute-force method to calculate the ARCP leads to unacceptable calculation time even if it is pre-computed. As a result, we propose a new grouping heuristic algorithm for targets heterogeneous bit-split string matching architectures. The algorithm uses specific measurements to estimate the correlation between strings. Correlation values play a key role in how strings are mapped to groups. This can allocate resources more efficiently, further reducing memory usage.

3.3. Pattern classification

To reduce the value of W_{PMV} mentioned in Section 3.2, we classify the rule strings in the original rule set using the unique and non-unique patterns defined in [26]. Kim H. et al. [9] propose a memory-efficient DFA-based string matching scheme that reduces the memory requirements by not storing the matching vectors. We extend this classification concept to the bit-split string. In the proposed scheme, all target patterns are categorized into two groups - a set with non-unique patterns and a set with unique pattern. For the set with non-unique patterns, the bit-split string matching technique with PMVs is applied. In contrast, for the set of unique patterns, only a single pattern is matched in each state. For the set of unique patterns, because the bit-split string matchers store only the pattern matching index, the memory requirement can be reduced.

3.4. Heuristic algorithm for dividing subgroups

If the length of each rule string is not the same, the number of states in the bit-split FSM is not the same. Allocating the same memory space to all FSMs will leave some memory units partially unused. Experimental evaluation of related work shows that these unused memory spaces (basically wasted) constitute an important part of the overall memory consumption. Existing algorithms also tried to alleviate this problem. Their goal is to reduce the total number of states required in FSMs by maximizing the common prefix shared between each group of strings. To do this, these methods sort the strings based on the prefix and then assign the strings to the appropriate groups. Such solutions can reduce the portion of the wasted space, but strings with the same prefix are not necessarily having exactly the same length, which will inevitably lead to wasted space. Other algorithms introduced a pattern grouping algorithm that uses the average length of the strings as crucial factor in grouping. The goal of our algorithm is to balance the number of target strings mapped onto each TILE. To do it, this method attempts to make the average length of the strings of each subgroup as close as possible to the overall average length of the strings. Although the above mentioned algorithm takes into account the length of the string, the most fundamental reason that really affects memory usage is the number of output states of the bit-split FSM.

We introduce and target heterogeneous architectures, where each block can have a different number of memory resources and configurations. This makes resource allocation more efficient, which significantly reduces overall memory usage. Aiming at the heterogeneous bit segmentation string matching architecture, a new string grouping heuristic algorithm is proposed. The algorithm uses specific measurements to estimate the correlation between strings. Correlation values play a key role in deciding how to map strings to each group. The algorithm makes decisions using precise string-related information.

Similar to the k-means algorithm, our proposed algorithm divides the target string set into specified subsets. The main idea of the algorithm is to let the target string in a category have the least S_{num} when converting to the final bit-split FSM. The algorithm mainly two steps: selecting sub-set sample points; and dividing rule set according to sample points.

In the first phase, a common character-level prefix between two target strings can reduce the total number of states in the FSM. Mapping strings with the longest common prefix to the same group can reduce memory consumption. In the bit-splitting method, the bit-level prefix is very important, because each FSM processes one or several bits of each character, and two different characters may have up to 7 common bits. Therefore, the common prefix of each parallel finite state machine should be measured; separately. First, we define a new "distance" to measure the relationship between two rule strings. The following formula is defined to as a measure for distance), d_{ij}, that estimates the benefit of mapping the i^{th} and j^{th} strings onto the same group:

$$d_{ij}=\frac{S_{num}\left(string_i\ ,\ string_j\right)}{S_{num}(string_i)+S_{num}\left(string_j\right)} \tag{6}$$

where $S_{\text{num}}(\,)$ is a function that calculates the total number of nodes required to represent the given strings inside one FSM. The higher number of common states in two strings in bit-level FSM, the smaller the numerator of the above formula, and the smaller the final "distance" value. That indicates a closer distance between the two characters. When the algorithm starts, the first string can be assigned to the first group. *DistanceSet* is a 2D array where *DistanceSet(i,j)* indicates the index of the j^{th} string assigned to the i^{th} group. Then the distance between first string and the rest of the strings is measured using d_{ij}. The results are stored in the first row of a 2D vector called *Distance_Vector*. Then, if more than two groups are requested, the algorithm considers the string that is the least correlated to those already assigned to the first group. The selected string is then assigned to the second group. In a similar way, the string with the smallest correlation distance from all previously assigned strings will be selected as the subsequent group. This process continues until the first members of all groups are identified. Estimation phase can be illustrated in line 1 to line 8 of Table 1 that highlights the pseudo code. The complete code is given in [27].

Table 1. Heuristic algorithm for dividing subgroups

```
Function RC(rule classification)
RC(Target rule Set，Number of Groups)
--------------------------------------------------------------
  # Estimation phase
  1    N= number of rule string in Target rule Set
  2    G= Number of Groups
  3    RC(1,1) = 1
  4    FOR i IN RANGE (1, G-1) :
  5        FOR j IN RANGE (1, N):
  6            $d_{ij} = d_{RC(i,1)j}$
  7        Find the rule R that has the minimum distance
                        with all already assigned rules  // store in DistanceSet
  8        RC(i+1,1) = index of R
  # Growing phase
      # In each string growing iteration, one of the ungrouped strings is selected and assigned to
      a group.
  9    FOR i IN RANGE (S+1, N):
  10       Find the rule RR that has largest difference in distance between the two
                                             most highly correlated groups
  11       Find the class n that has the minimum distance with RR
  12       RC(n, end+1) = RR
  13       Update distance values of class n using $D_{ij}$
                  (DistanceSet(n,j),1<j<N) using Eq.(6)
```

In the second phase, the algorithm maps the remaining strings onto the groups in an iterative manner. In each string growing iteration, one of the ungrouped strings is selected and assigned to a group. The selection order has a great impact on the results. The proposed algorithm defines and uses a criterion to determine the string placement order. The criterion is as follows: the highest priority is given to the string that has the largest difference in Distance value with the closest group and with the other groups. Therefore, the assigned string is part of that group. The elements related to the assigned string are removed from the *Distance_Vector* since that string is not ungrouped anymore. In the meantime, the distance values of all remaining elements with the enlarged group are updated. This process continues until all strings have been grouped. The following formula is used for updating the Element-Group Distance values:

$$D_{ij}=\frac{S_{num}\left(group_i,\ string_j\right)}{S_{num}(group_i)+S_{num}\left(string_j\right)} \tag{7}$$

where $group_i$ is the set of all strings already assigned to the i^{th} group and, thus, $S_{\text{num}}(group_i)$ represents the total number of states required to represent all strings in $group_i$. Growing phase can be illustrated in line9 to line13 of Table1 pseudo code.

4. Experimental results

The performance of HARD method was evaluated using a simulator implemented in Python. The platform used in the experiment is a PC system with 3.1GHz Intel Core i5-2400 2 CPU and 8GB memory. We choose the following three popular rule sets as target rules, as shown in Table 2.

Table 2. The rule sets selected for the experiment.

Name	NUM	SIZE(B)	AL*	SD#
Snort v2.8.spyware	2299	26103	11.4	8.1
Snort v2.8.web-client	1657	67527	40.8	22.8
ClamAV	28786	1921305	63.2	40.8
suricata	2974	91216	33.4	19.6

*AL - True Average Length #SD – Standard Deviation

Snort [28] is an open source intrusion prevention system capable of real-time traffic analysis and packet logging. ClamAV [29] is an open source antivirus engine for detecting Trojans, viruses, malware & other malicious threats. Suricata [30] inspects the network traffic using a powerful and extensive rules and signature language, and has powerful scripting support for detection of complex threats. The rule base we have used has a various length. The variation of string length in the rules can be seen from the value of the standard deviation.

4.1. Effect of parameter selection

HARD has an important parameter, namely *GroupNum*. It affects the results in different ways, and it is crucial to select an appropriate value for it. We randomly selected 200, 1000, 2800 rule strings from the *ClamAV* library for different experiments. Each time *GroupNum* takes a value of 1, 2, 10, 30, 50, 100 or 200. The experimental results are shown in the Fig. 4. The results show that when the number of group grows in a narrow lower range, the memory saving ratio (changed memory space / original memory usage) is significant; after that small range, the saving ratio is flattened out. The tread can be further explained in Fig. 5. When the value of *GroupNum* changes from 1 to 2, it can save about 35% of the memory. From 2 to 10, the memory saving ratio is as high as 60%. As the value of *GroupNum* increases, the saving ratio slows down. When the number of groups is about 50, the effect of grouping on memory saving is already weak, which indicates that memory savings are approaching the limit when there are about 50 groups. After that, the saving effect is not obvious anymore. In the extreme case, the number of groups is set to the number of rules, then the respective memory is applied according to the number of bit-split FSM output states corresponding to each string. Although there is no memory waste in this case, the time overhead will increase dramatically since the complexity of the grouping algorithm is $O(n^2)$. As a trade-off, the optimal value for *GroupNum* is taken around 50. By the way, although the packet algorithm time overhead complexity is $O(n^2)$, it does not add too much burden to the system since the rule database is usually only updated every few days or weeks for such intrusion detection systems.

4.2. Comparison to existing methods

We tested our algorithm on four different rules data sets, and at the same time we compared the performance of the algorithms to those in [8] and [9]. In order to reduce the impact of the number of rules in the same database on the algorithm test results, we randomly selected (200, 600, 1000, 1400, 2200, 2800 and 4000) rules in the same database for testing. There are two important adjustable parameters in the comparison work: p, the bit-width of a PMV which defines the maximum number of strings that can be mapped onto one group, and s, the number of states in each FSM. The best parameter values are found through different values of these parameters. Figure 6 illustrates the results achieved by the best parameter values for the existing approaches. $p = \lceil \log_2 gL \rceil$, g means number of bits in a partial match vector. L

means number of characters per string on average, it can be derived from Table 2. It is worth mentioning that in these experiments the parameter *GroupNum* is set to 50.

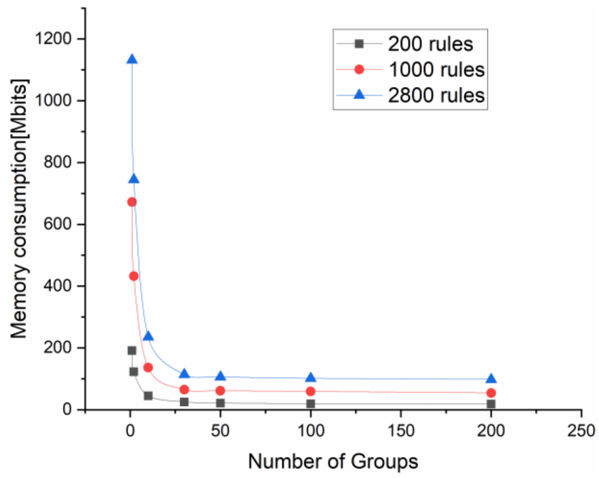

Fig. 4. Memory consumption under different groups. When the number of groups is about 50, the effect of grouping on memory saving is already weak, which indicates that memory savings are approaching the limit when there are about 50 groups

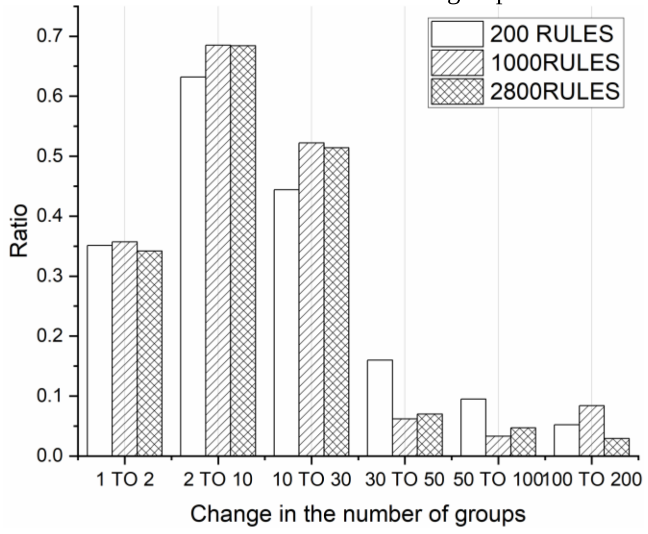

Fig. 5. Memory saving ratio during grouping

Fig. 6 shows the performance comparison of the HARD and the algorithms in [9] and [11]. It can be clearly seen that HARD can effectively save memory. First of all, it is obvious, that the bit-split AC algorithm requires the most memory in all the tested cases. This is simply because the bit-split FSM uses a uniform memory size and a lot of memory space is wasted. In addition, from the comparison of results from step2 and step1+step2 in each sub-graph, it can be seen that the first step of the algorithm plays an important role in the overall performance of the algorithm. Step1+step2 further improves the performance for different data sets. Since the classification in step1 can effectively reduce the use of W_{PMV}. It is worth noting that step1 in Fig. 6 (d) does not show any significant memory saving because almost all rule strings in its rule set belong to non-unique pattern. Secondly, the performance of our method is more stable as the number of rules increases. This is due to the fact that our algorithm can reduce the number of states on the FSM and the space for PMV. In contrast, the method used in [9] and [11] mainly tried to reduce the number of states on the FSM, and not on PMV space. From Fig. 6, it can also be seen that HARD performs much better on *ClamAV* than other algorithms. This is because the length of the rule strings in the *ClamAV* varies a lot. From the standard deviation of each database, we can see that the algorithm performs better on datasets with larger standard deviations.

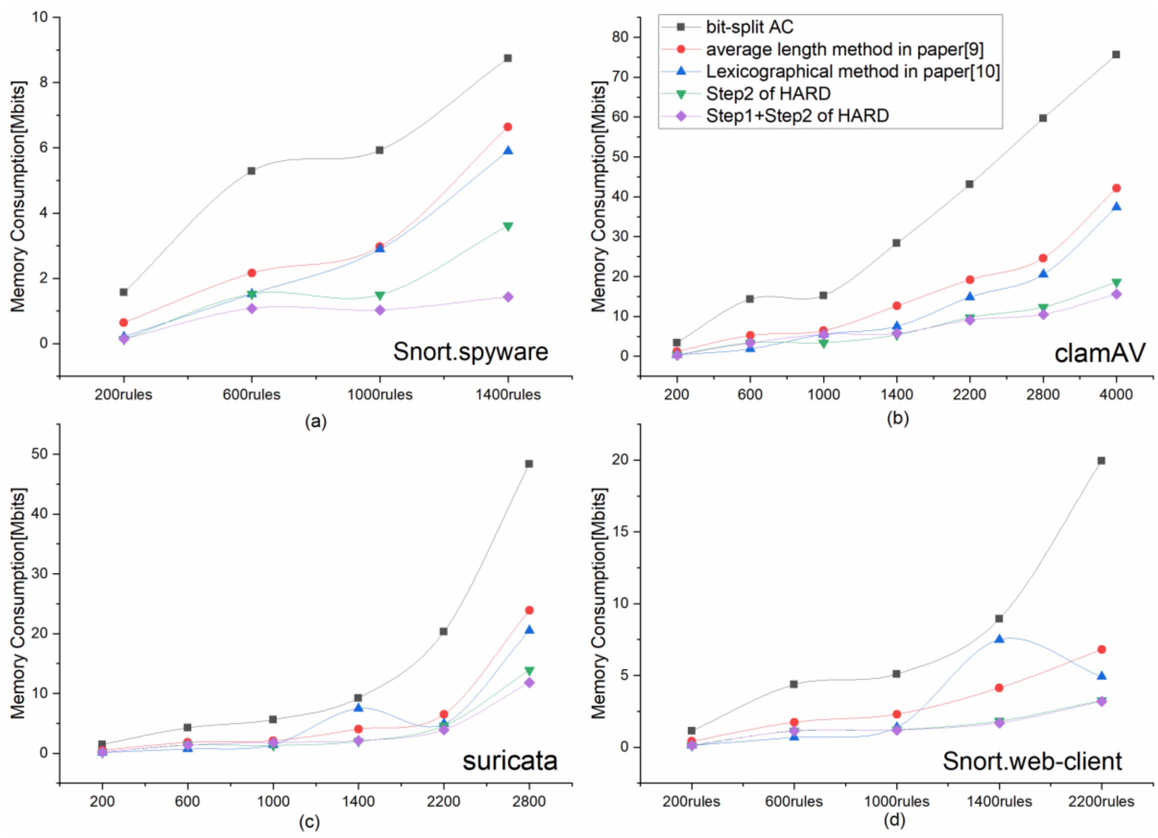

Fig. 6. Memory comparison of different algorithms on different rule sets. The more the number of rules in the rule set, the more obvious the effect of HARD. When the number of rules is above 4000, HARD can save nearly 50% of memory consumption compared to the previous bit-split string matching methods.

Fig. 7 shows the actual memory consumption and the waste of space due to structural reasons for different tested cases. We can see that the actual memory size requirement for different algorithms is actually not much different (since it is determined by the rules size itself). However, in the wasted space portion, each algorithm is completely different since different algorithms use different data organization methods and data structures. HARD applies for space on demand, and there is almost no space waste. When processing rules with different lengths in other algorithms, they can only apply for space based on the longest character, causing a certain amount of space waste. Lexicographical [9] sorting aims to reduce the overall number of states required in FSMs by maximizing the shared common prefixes among the strings within each group. For this purpose, these approaches sort the strings based on their prefixes and then assign the strings to the groups in order. Although the method in [9] is optimized for organization on common subsequence, the size of FSM in each group is determined by the rule with the largest number of states. Since all state machines use the same size of storage space, it inevitably leads to larger memory waste. Although the method in [11] tries to place each rule as much as possible to groups with similar average lengths, it is impossible to have the same length of all rules, so there is still a certain amount of wasted space. HARD reduces the consumption from two aspects. First, the state in the final rule within a group is minimized in the bit-split FSM, which essentially reduces the actual consumption. Secondly, the bit-split FSM size in each TILE is allowed to be different. This will effectively reduce the wasted space. Experimental results show that increasing the number of groups can also save more memory, which is more significant in scenarios with a larger number of rule string sets.

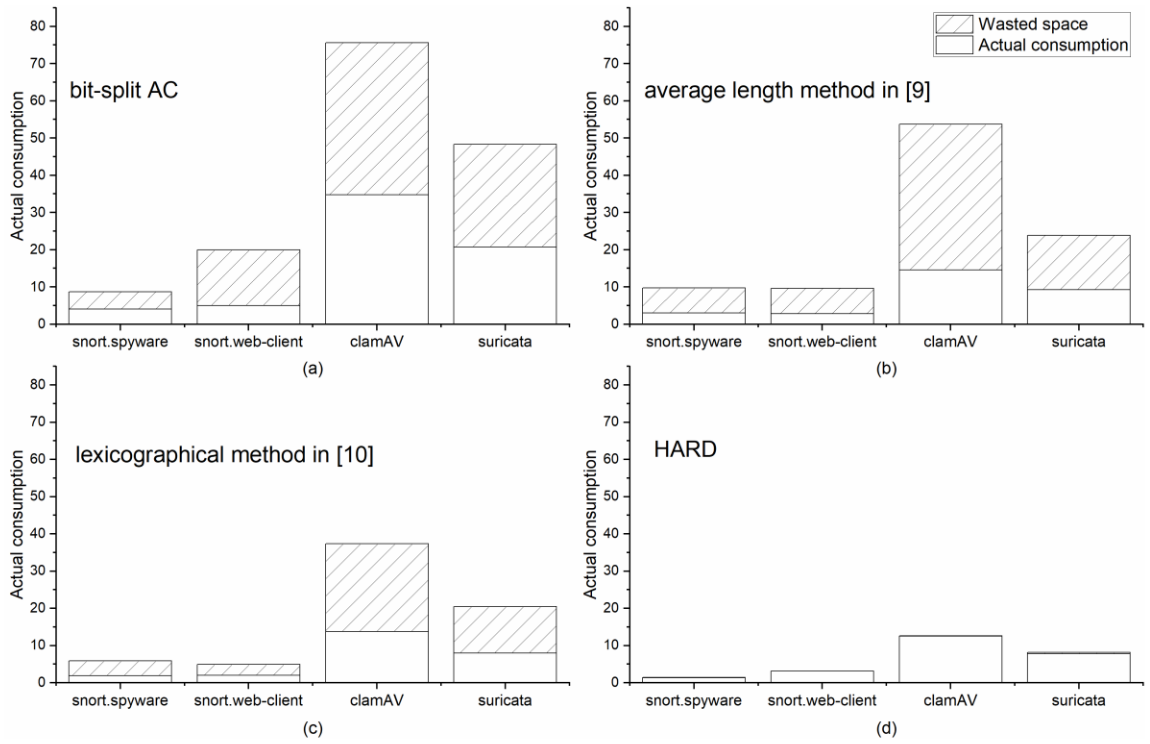

Fig. 7. Comparison of wasted space and actual consumption. HARD requests space on demand, and there is almost no space waste. Other algorithms have different lengths of rule characters, and can only apply for space according to the longest characters, resulting in a certain amount of space waste.

5. Discussion

We proposed a method HARD in order to save memory. The HARD first uses the uniqueness of the target pattern to classify all target rule characters into two categories. Then applies a method to estimate the distance between strings in unique pattern category. The distance is then used to find a preferred placement of each rule into different groups. Furthermore, each group will be processed on different sizes of FSM. The HARD was assessed through considering several test cases. The results demonstrate that the memory saving increases with the increase of the number of groups. The performance of HARD becomes more obvious as the set of strings becomes larger. The drawback of using a larger number of groups is that more hardware circuit are required.

Acknowledgements. The authors would like to thank all the sponsors. This research was supported by the Open-end fund project of Science and Technology on Communication Networks Laboratory (No. HHX20641X004).

References

[1] LE H., PRASANNA V. K., *A memory-efficient and modular approach for large-scale string pattern matching*, Computers, IEEE Transactions on, **62**, pp. 844–857, 2013.

[2] HUA N., SONG H., LAKSHMAN T., *Variable-stride multi-pattern matching for scalable deep packet inspection*. Proceedings of INFOCOM 2009, IEEE, pp. 415–423, 2009.

[3] PAO D., LIN W., LIU B., *A memory-efficient pipelined implementation of the Aho-Corasick string-matching algorithm*. ACM Transactions on Architecture and Code Optimization, **7**, pp. 10, 2010.

[4] YUN S., *An efficient TCAM-based implementation of multi-pattern matching using covered state encoding*. IEEE Transactions on Computers, **61**, pp. 213–221, 2012.

[5] AHO A. V., CORASICK M. J., *Efficient string matching: an aid to bibliographic search*. Communications of ACM, **18, 6,** pp. 333-340, 1975.

[6] KHAN Z. A., PATERIYA R. K., *Multiple pattern string matching methodologies: a comparative analysis*. International Journal of Scientific and Research Publications, **2, 7**, pp. 1-7, 2012.

[7] JONY A. I., *Analysis of multiple string pattern matching algorithms*. International Journal of Advanced Computer Science and Information Technology, **3, 2**, pp. 344-353, 2014.

[8] TAN L., BROTHERTON B., SHERWOOD T., *Bit-split string-matching engines for intrusion detection and prevention*. ACM Transactions on Architecture and Code Optimization, **3**, pp. 3–34, 2006.

[9] KIM H., CHOI K. I., CHOI S. I., *A memory-efficient deterministic finite automaton-based bit-split string matching scheme using pattern uniqueness in deep packet inspection*, Plos One, **10**, 2015.

[10] KIN H., KANG S., *A pattern group partitioning for parallel string matching using a pattern grouping metric*. IEEE Communications Letters, **14, 9**, pp. 878-880, 2010.

[11] BOYER R. S., MOORE J. S., *A fast string searching algorithm*. Communications of the ACM, **20**, pp. 762–772, 1977.

[12] ABDULLAH A., NAZIR A., SENAPAN M., MENG S. S., KARUPPIAH E., *Fast multi-keyword range search using GPU*. In: GPU Computing and Applications. Springer, pp. 259–274, 2015.

[13] PURCARU C., PRECUP R.-E., IERCAN D., FEDOROVICI L.-O., DRAGAN F., *Optimal robot path planning using gravitational search algorithm*. International Journal of Artificial Intelligence, **10**, pp. 1-20, 2013.

[14] MLS K., PUHEIM M., VASCAK J., CIMLER R., *Interactive evolutionary optimization of fuzzy cognitive maps*. Neurocomputing, **232**, pp. 58-68, 2016.

[15] ABED ALGUNI B., *Island-based Cuckoo Search with Highly Disruptive Polynomial Mutation*. International Journal of Artificial Intelligence, **17**, pp. 57-82, 2019.

[16] PRECUP R.-E., DAVID R.-C., PETRIU E. M., SZEDLAK-STINEAN A.-I., BOJAN-DRAGOS C.-A., *Grey wolf optimizer-based approach to the tuning of PI-fuzzy controllers with a reduced process parametric sensitivity*, IFAC PapersOnLine, **49**, pp. 55-60, 2016.

[17] SOURDIS I., PNEVMATIKATOS D. N., VASSILIADIS S., *Scalable multigigabit pattern matching for packet inspection*. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, **16**, pp. 156–166, 2008.

[18] KEFU X., DEYU Q., ZHENGPING Q., WEIPING Z., *Fast dynamic pattern matching for deep packet inspection*. Proceedings of ICNSC 2008, IEEE, pp. 802–807, 2008.

[19] AHO A. V., CORASICK M. J., *Efficient string matching: an aid to bibliographic search*. Communications of the ACM, **18**, pp. 333–340, 1975.

[20] SOURDIS I., PNEVMATIKATOS D., *Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system*. In: Field Programmable Logic and Application. Springer, pp. 880–889, 2003.

[21] CHEN C. C., WANG S.D., *An efficient multi-character transition string-matching engine based on the Aho-Corasick algorithm*. ACM Transactions on Architecture and Code Optimization, **10**, pp. 25, 2013.

[22] YU .F., KATZ R. H., LAKSHMAN T. V., *Gigabit rate packet pattern-matching using TCAM*. In: Proceedings of 12th IEEE International Conference on Network Protocols, IEEE, pp. 174–183, 2004.

[23] HUANG K., ZHANG D., QIN Z., *Accelerating the bit-split string matching algorithm using Bloom filters*. Computer Communications, **33**, pp. 1785–1794, 2010.

[24] CHEN C., QIN Z., *A bit-split byte-parallel string matching architecture*. In: Proceedings of International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, IEEE, pp. 214–217, 2009.

[25] LIN P. C., LIN Y. D., LEE T., LAI Y., *Using string matching for deep packet inspection*. IEEE Computer, **41**, pp. 23–28, 2008.

[26] SAHNI S., GONZALEZ T., *P-complete approximation problems*, J. Assoc. Comput. Mach., **23**, pp. 555–565, 1976.

[27] https://github.com/xjtuleeson/HARD

[28] Snort. Snort v2.8. web-client dataset [DS/OL].[2020-01-10]. https://www.snort.org/downloads/#ruledownloads

[29] ClamAV. ClamAV dataset [DS/OL]. [2020-01-10]. http://www.clamav.net/downloads

[30] Suricata. Suricata dataset [DS/OL]. [2020-01-10]. https://suricata-ids.org/download/