Multimedia Tools and Applications (2020) 79:16245–16263
https://doi.org/10.1007/s11042-019-07822-8

XWM: a high-speed matching algorithm for large-scale URL rules in wireless surveillance applications

Shuzhuang Zhang[1] · Yanbin Sun[2] · Fanzhi Meng[3] · Yunsheng Fu[3] · Bowei Jia[1] · Zhigang Wu[1]

Received: 30 December 2018 / Revised: 29 April 2019 / Accepted: 21 May 2019 /
Published online: 10 June 2019
© The Author(s) 2019

Abstract
Large-scale high-speed URL matching is a key operation in many network security systems and surveillance applications in Wireless Sensor Networks. Classic string matching algorithms are unsuitable for large-scale URL filtering due to speed or memory consumption. This paper proposes an extend Wu-Manber algorithm (XWM) which takes advantage of the encoding characteristics of the URL greatly to improve the matching performance of the algorithm. It first adopts the pattern string window selection method to optimize Wu-Manber's hash process, and then combines hash tables and associative containers to optimize the string comparison process. The experimental results on actual 10 million patterns show that XWM can achieve speeds that are twice as fast as traditional algorithms, especially when the shortest pattern string length is longer, it is more advantageous.

Keywords Multi-string matching · URL matching · Wu-Manber algorithm

1 Introduction

The Hypertext Transfer Protocol (HTTP) is one of the most widely used internet protocols at present. In addition to the traditional desktop applications, many mobile applications use the HTTP protocol for data transfer [17]. The URL (uniform resource locator) is the most important component of the HTTP protocol, identifying the location of the requested resource. Filtering harmful URLs can effectively control and manage access to illegal and harmful information. Detection of harmful URL information in real-time network traffic is an important

✉ Yanbin Sun
sunyanbin@gzhu.edu.cn

[1] Institute of Network Technology, Beijing University of Posts and Telecommunications, Beijing 100786, China

[2] Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou 510006, China

[3] Institute of Computer Application, China Academy of Engineer Physics, Mianyang 621900, China

element of current security systems, it has a wide range of applications in the field of network information security, including traditional network intrusion detection/defense systems (IDS/IPS) and surveillance applications in Wireless Sensor Networks [5, 6].

However, due to a large number of URL rules, traditional string matching algorithms cannot successfully filter tens of millions of URLs in real time. Measures need to be taken to improve and optimize the characteristics of URLs for effective string matching. Based on the classic Wu-Manber multi-pattern string matching algorithm [18], we propose a new XMW algorithm that augments Wu-Manber with the characteristics of URL data to improve its matching performance, especially for longer lengths of the shortest pattern string. The second section of this paper introduces related work for large-scale URL pattern string matching. The third section describes the improvement measures and algorithms proposed in this paper in detail. The fourth section experimentally evaluates the improved algorithm against other string matching algorithms. The fifth section is the conclusion of this article.

2 Related work

The surveillance application using wireless networks acquires real-time and accurate multimedia information conveniently, and it is widely used in IoTs [10], IoVs [16], and so on. Most of wireless network researches focus on the data transport and management [13, 14], the private-preserving [3, 4, 21] and network attack [12, 15], which are also suitable for wireless multimedia surveillance networks. In addition, for the multimedia surveillance application, the illegal and harmful multimedia data should be prevented according to the multimedia content or the URL of content. Our work focuses on the latter method, i.e., filter harmful multimedia URLs via URL matching.

URL matching is a typical application of pattern string matching. However, the widespread use of the HTTP protocol means that the number of rule sets for matching URLs is huge, reaching tens of millions of patterns of visible characters. There are two primary matching methods for URLs. One uses classic pattern string matching methods directly. The other improves classic pattern string matching methods by mining the characteristics of the URL.

There are three general methods for classical pattern-based string matching that offer possibilities for our purposes: automata, hash tables, and bit parallel.

Typical representatives of automata-based algorithms such as the Aho-Corasick automata (AC algorithm) [1] and Factor oracle automata (SBOM algorithm) [2]. The matching performance of this type of algorithm is stable and unaffected by the length of the pattern string and character distribution. The time complexity of such algorithms is proportional to the length of the text string to be matched. When applied to large-scale URL string matching, this type of algorithm consumes a large amount of memory for automata storage. In response to this issue, Xiong G et al. [19] proposed a hybrid automaton construction algorithm based on data access and using statistical strategies based on the AC algorithm. The result of Real world testing showed that it was only able to make a reduction in memory use about 5%.

Hash-based multi-pattern string matching algorithms use hash tables and add character block technology to increase the possibility that the text string and the pattern string do not match, thereby improve the chance of jumping. The Wu-Manber algorithm [18] is the typical representative of this type of algorithm, achieving better performance with 100,000 random strings. However, URL strings have semantic features; the distribution of characters is not random. Therefore, the Wu-Manber algorithm offers weak performance due to a high rate of

hash collisions when matching URLs. Zhang P et al. [23] proposed the HashTrie algorithm, which uses recursive hashing technique to store the processed pattern string set information in a bit vector and a rank operation for quick verification. This algorithm uses 0.4% of the memory overhead of the AC algorithm but offers actual matching performance that is only about half that of the AC algorithm. The performance makes it unsuitable for high-speed matching applications.

Bit parallel algorithms simulate the matching process of automata by using bit vectors. Operations on the bit vectors replace the state jumps of the automata, and execute in parallel using a machine word. The representatives of this kind of algorithms are the shift-and and shift-or algorithm [8, 9]. However, the machine word length limits this type of algorithm, which is effective only with small-scale pattern strings. Salmela L et al. used *q-gram* technology to expand the shift-or algorithm [11] (SOG). This approach uses *q-gram* technology to serialize multiple-pattern strings into a simple single pattern string and then uses fast single-pattern string matching technology to filter text that cannot be matched quickly. This technique achieves better results with 10,000 to 100,000 pattern strings but is still unsuitable for large-scale matching.

Research into large-scale URL matching has focused primarily on improving the classic algorithms and enhancing matching performance by making full use of the character sets and coding characteristics of URLs. Liu YB et al. [7] proposed a filtering-based algorithm based on the classic SOG algorithm (SOGOPT) for large-scale URLS flitting. It combines two optimizations: pattern string window selection and packet reduction, which greatly improve the matching performance of the algorithm. However, this algorithm is limited by the system's machine word length and the shortest pattern string length. When the machine word length is shorter or the shortest pattern string length of the pattern set is longer, the number of pattern strings that can be searched concurrently is reduced, which reduces the performance of the algorithm. Yuan Z et al. [22] proposed a multi-pattern matching algorithm which employs Two-phase hash, Finite state machine and Double-array storage to eliminate the performance bottleneck of blacklist filter (TFD). However, since the trie data structure is used in the algorithm, the memory consumption of the algorithm and the double-array AC automata have a considerable magnitude, which limits the algorithm's application. Xu DL [20] proposed partitioning URLs by "/" and ".". This algorithm achieved higher matching performance based on URLs filtering. However, this method only supports block URL prefix matching and does not support substring matching, which limits the scope of application.

In summary, scholars have researched large-scale URL matching in recent years, but there is still a lack of effective algorithms. The following section introduces a new algorithm for more effective URL matching of multiple patterns.

3 The XWM algorithm

Based on the Wu-Manber algorithm, our algorithm optimizes the characteristics of URL data and proposes an algorithm called XWM (eXtend Wu-Manber) for large-scale URL pattern string matching. The XWM algorithm improves matching performance with three optimizations methods.

The first optimization is the adoption of the pattern string window selection technique proposed by Liu YB et al. [7]. This technique selects a unique and representative window for

each pattern string to represent each pattern string uniquely. This significantly reduces the probability of a hashing function placing multiple pattern strings in the same bucket.

The second optimization is the adoption of a two-phase hash. Our algorithm uses two compressed hash tables to store the jump values for the matching process. It can significantly reduce memory usage while ensuring uniform hashing.

The third optimization is the use of associative containers to organize conflicting mode strings and remove the need for the prefix table in the Wu-Manber algorithm. The associative container locates key values quickly and significantly reduces the number of comparisons at the time of verification.

This section analyzes the shortcomings of the Wu-Manber algorithm in large-scale URL matching in Section 3.1. Then three optimization techniques used in this paper are introduced in Sections 3.2, Section 3.3 and Section 3.4, respectively. Finally, the preprocessing and matching process of the proposed algorithm is given in Section 3.5.

3.1 Analysis of the Wu-Manber algorithm

The Wu-Manber algorithm is a classical multi-pattern string matching algorithm proposed by Sun Wu in 1994. The algorithm uses the idea of a “Bad Character” to jump. In the preprocessing phase, three hash tables are created: the *shift* table, the *hash* table, and the *prefix* table. When scanning a text string, the shift table determines the number of characters to jump backwards based on the read string. The hash table stores the pattern strings with the same tail block character hash value. The prefix table stores the first block character hash value of the pattern string with the same tail block character hash value. In the matching process, if the current text string's shift value is zero, it indicates that a match is possible, and further verification is needed. In this case, the prefixes of the same pattern strings of the tail block are compared to the ones in the prefix table. If those match, the algorithm evaluates them one by one in the hash table for the given tail block to find a match. Figure 1 shows the shift table, the hash table, and the prefix table of patterns {abcde, bcbde, adcab}.

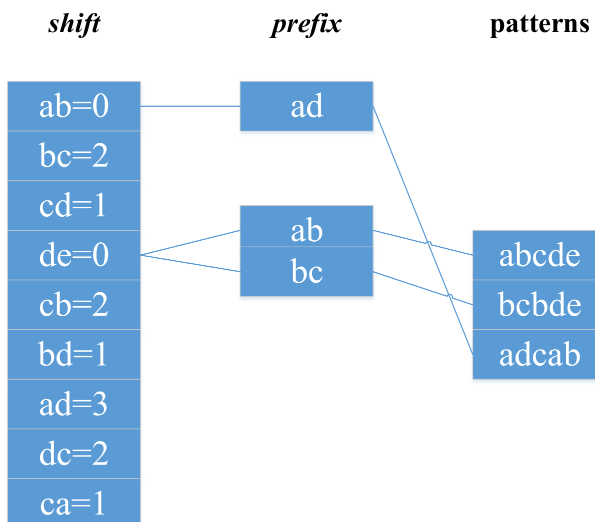

Fig. 1 the *shift* table, the *hash* table, and the *prefix* table of patterns {abcde, bcbde, adcab}

When the Wu-Manber algorithm is directly applied to large-scale URL matching, the matching performance is low, mainly for two reasons. The first reason is the severity of hash collisions. The Wu-Manber algorithm uses 2 to 3 bytes as character block, which is appropriate for matching small and medium hash strings when hashing. Hash collisions increase with the size of the pattern strings. Additionally, URLs have many common prefixes and suffixes (for example, "www", "com", "cn", etc.). The Wu-Manber algorithm uses the leftmost m strings of the pattern string as the matching window. Since many pattern strings have the same matching windows, hash collisions become serious.

The second reason is that accurate calibration takes a long time. In the course of matching, the Wu-Manber algorithm enters the precise verification phase when the shift value is zero. Since the conflicting pattern string is stored in the hash table as a single-linked list, it is necessary to traverse the single-linked list to determine a match when verifying. As just noted, URLs often have the same prefixes, resulting in a conflicting linked list with the same prefixes is very long. Thus, it requires significant CPU time when traversing the list to exact matches.

3.2 Pattern string window selection

Liu YB et al. [7] proposed the pattern string window selection technique for optimizing the SOG algorithm. This optimization technique is suitable to the Wu-Manber algorithm as well. Here we select the length m of the shortest pattern string as the matching window size as well as the original Wu-Manber algorithm. However, we cannot use the leftmost m characters of each pattern string as the window of each pattern string, since URLs have an uneven character distribution as a result of common prefixes and suffixes, which may result in a large number of pattern strings having the same matching window. The pattern string window selection technique selects a unique and representative window for each pattern string. The uniqueness reduces hash collisions. Our algorithm uses this technique.

For example, consider pattern string set P = *{google.com, google.com.hk, google.com.tw, google.com.jp, google.com.tr}*. If the leftmost 10 characters are used as the window for all pattern strings, all pattern string sets will have the same matching window *google.com*. All five pattern strings in the set hash into the same bucket regardless of the hash function. Using the window selection technique to process the pattern string set yields a matching window for each pattern string as shown in Table 1. Each pattern string has a different matching window, which reduces the probability of hash collisions.

3.3 Two-phase hash

The original Wu-Manber algorithm selects the leftmost B characters, usually two or three of each pattern string as a hash block. The algorithm uses $B = 2$ when the pattern string size is small and $B = 3$ when the pattern string size is large. More generally, $B = log_{|\Sigma|}(2 * m * r)$,

Table 1 An example of window selection for each string

Pattern string	Window
google.com	*google.com*
google.com.hk	*ogle.com.h*
google.com.tw	*gle.com.tw*
google.com.jp	*ogle.com.j*
google.com.tr	*gle.com.tr*

where $|\Sigma|$ is the size of the character set (generally taken as 256, the size of the extended ASCII table), and r is the number of pattern strings. The hash table size depends on B. Larger values of B improve the matching performance of the algorithm but seriously increase the memory usage of the hash table. For example, when $B = 4$, the hash table size is 2^{4*8}, which is 4GB.

Our algorithm does not use a constant B but uses the formula $B = \alpha m$ to determine the size of B dynamically, where α $(0 < \alpha < 1)$ is a factor that determines the size of B. Our algorithm uses two compressed hash tables to improve the matching performance of the algorithm while keeping the memory consumption low. We refer to these two hash tables as the *shift* table and the *hash* table. The *shift* table and the *hash* table have 2^m and 2^n entries and use hash functions h_1 and h_2, all respectively. Function h_1 maps a character block with length B into a value of m binary bits, and h_2 maps a character block with length B into a value of n binary bits. We use the *shift* table to determine the number of characters to skip when scanning the text string. The *hash* table organizes the pattern strings with tail block characters that hash to the same value.

In fact, if the currently validated character block never appears in the other pattern strings or the right end of the pattern string matching window, the current matching sliding window can be moved back a greater distance. For this reason, we add a skip value to the hash table for accurate verification of backwards jumps. In the matching process, h_1 first calculates the last B character strings in the current matching window. If the corresponding value in the shift table is not zero, the backward jump is performed. Alternatively, a value of zero indicates that there may be a match. At this time, h_2 calculate the hash value of the last B character strings in the current matching window and checks the corresponding table entries in the hash table. If there are conflicting pattern strings, the algorithm performs an exact verification and uses the *skip* value after verification to shift (jump) the matching window of the current text backwards. Otherwise, the current match window is jumped backward according to the skip value in the hash table entry.

3.4 Using associative containers to organize conflicting nodes

In the Wu-Manber algorithm, potential matches found with the prefix table require verification to determine whether an identical pattern string matches by traversing the corresponding conflicting linked list of the hash table. The one-by-one string comparison process is extremely time-consuming. To reduce the number of verifications and make full use of the windows described in Section 3.2, we use an associative container to organize the lists for conflicting nodes in the hash table and omit the prefix table of the Wu-Manber algorithm.

An associative container is a type of data structure that stores and retrieves elements efficiently through key values, typically using a balanced binary tree or hash table. We used both forms to implement the XWM-Tree and XWM-Hash algorithms for the sake of comparison (see Section 4). Key value construction is central to the use of associative containers. The prefix table in Wu-Manber stores the character hash value of the first block of the pattern strings with the same tail block character hash value. To replace the prefix table, we use the prefix's hash value of each pattern string matching window as the key value of each pattern string. Thus, the associative container can completely replace the function of the prefix table without affecting the behavior of the algorithm. Since only the pattern strings with the same key value need verification using the container, and the container facilitates a fast search, the time required for verification decreases significantly.

3.5 Algorithm description

Our algorithm consists of two stages: a preprocessing phase and a scanning phase. The preprocessing phase performs the following steps. First, it determines the representative matching window for each pattern string using the window selection technique. Second, it generates the shift and hash tables according to the suffix character blocks of each pattern string matching window. Third, it calculates the key value according to the prefix string of the matching window and inserts each pattern string into the corresponding associative container in the hash table.

Figure 2 shows a diagram of preprocessing, with gray-shaded elements representing the matching window selected for each pattern string. Light gray indicates the portion used to calculate the tail block of the pattern string window for the shift and hash tables. Dark gray indicates the portion used to calculate the prefix strings of the pattern string window for the corresponding key value. The map field in the hash table stores a pointer to the associative container. With the pointer and the calculated key value, pattern strings can be inserted separately into the corresponding entry in the hash table.

The scanning phase begins after the preprocessing phase completes. Algorithm 1 presents a pseudo-code description of the scanning and matching process. The matching process needs to maintain a matching window of size *m*. Lines 3 through 12 use the suffix string of the current text matching window to calculate the shift value. If the shift value is not zero, the current text is skipped backwards; lines 13 through 18 deal with the case where the shift value is zero. At this time, a hash value is calculated using the suffix string of the matching window of the current text. If the associative container is not empty, the prefix *hash* value of the matching window used as the key value.

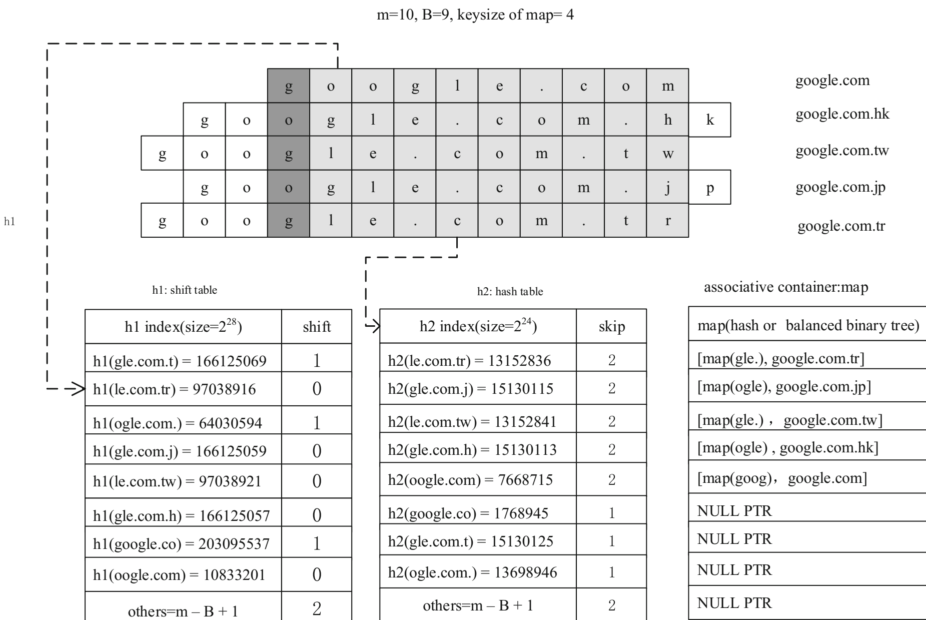

Fig. 2 Pattern string preprocessing diagram

Comparisons are performed by searching the associative containers with the key value. Line 19 is the process of jumping backwards after performing an exact check.

Algorithm 1: Matching process

---

input: *text (text represents the content* to be scanned)

1: *window*← *text* + *m* - *1*, *text_end*←*text* + *len*(*text*)

2: **while** *window* < *text_end* **do**

3: *hindex*← *h1*(*window*)

4: *shift_value*←*shift*[*hindex*]

5: **while** *shift_value*←0 **do**

6: *window*←*window* + *shift_value*

7: **if** *window* > *end* **then**

8: **return**

9: **end if**

10: *hindex*←*h1*(*window*)

11: *shift_value*← *shift*[*hindex*]

12: **end while**

13: *hindex*←*h2*[*window*]

14: *skip*←*hash*[*hindex*].*map*

15: **if** *hash*[*hindex*].*map* ≠ null **then**

16: *key*←*h2*(*window* - *B*)

17: find match *hash*[*hindex*].*map*[*key*]

18: **end if**

19: *window*←*window* + *skip*

20: **end while**

Algorithm 1 Scanning and matching the text

Compared with the original Wu-Manber algorithm, our methods first use hash table to replace the original shift table in Wu-Manber, which allows the algorithm to use a larger B and reduce the conflict in each hash bucket. Then use a heterogeneous hash table and associated container to replace the prefix table in Wu-Manber, in this way, our algorithm speeds up the matching process of string when encountering hash conflicts.

4 Experimental evaluation

In order to compare and explain the performance of the proposed algorithm, we selected a representative algorithm from each type of the classical matching algorithms according to their

Table 2 Comparison of the spatial complexity and time complexity in different algorithms

	XWM-HASH	XWM-TREE	AC	SOGOPT	TFD
The time complexity	$O(n)$	$O(n)$	$O(n)$	$O(n(1 + p'(r,G)logr)$	$O(n)$
The spatial complexity	$O(\|P\|)$	$O(\|P\|)$	$O(\|P\|)$	$O(w\|\sum\|^q)$	$O(\|P\|)$

principles to be used as different comparisons. In this way, the auto-based AC algorithm, HASH-based TFD algorithm and Bit parallel-based SOGOPT algorithm, which can also support large-scale rule sets, were selected and implemented.

We compared the speed and memory consumption of the two variants of our algorithm, XWM-Tree and XWM-Hash, with the SOGOPT algorithm (using 64-bit machine word length), the TFD algorithm, and the double-array AC algorithm (da_ac).

The spatial complexity and time complexity of these algorithms are shown in Table 2:

Where n is the length of the text to be matched, $|P|$ represents for the sum of the length of all the patterns, $|\sum|$ is the character set size, r is the number of pattern strings in the rule set P, G is the number of pattern string set packets in the SOG algorithm, and p' is the probability of entering the check in the SOG algorithm.

We also discussed the effect of the shortest pattern string length on the algorithm. We set $\alpha = 0.75$, $m = 28$, and $n = 24$ in the XWM-Tree and XWM-Hash algorithms.

4.1 Experimental data and experimental environment

We used a list of approximately 80 million URLs (15 GB) taken from a backbone router as our strings to test and extracted more than 10 million pattern strings from it for the pattern sets.

Our experimental hardware and software environment was as follows: Intel Xeon E5-2650 v3 CPU at 2.3GHz, 32 GB of memory, and the Red Hat Enterprise Linux Server release 7.0

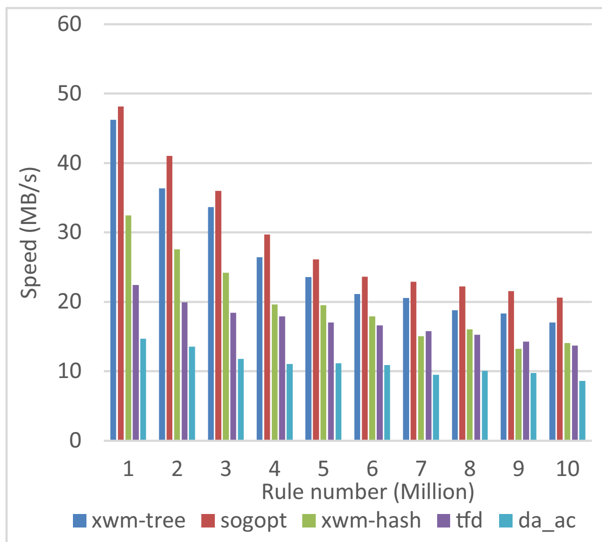

Fig. 3 Comparison of speeds of different algorithms when the shortest pattern string length is 6

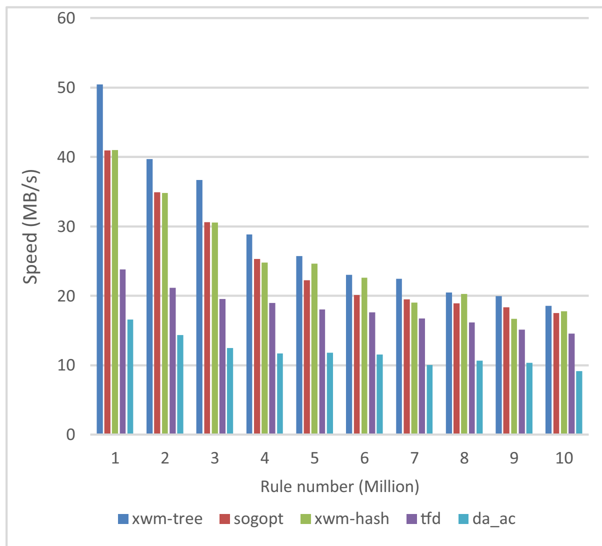

Fig. 4 Comparison of speeds of different algorithms when the shortest pattern string length is 8

(Maipo) operating system with kernel version is 3.10.0-123.el7.x86_64. All code was written in C++, compiled with g++ 4.8.2 using -O3 optimization during compilation. All programs run single-threaded.

4.2 Experimental results and analysis

Figures 3, 4, 5, 6, 7 and 8 show that both XWM-Tree and XWM-Hash had higher matching performance than other algorithms. When the shortest pattern string length is 8, XWM-Tree,

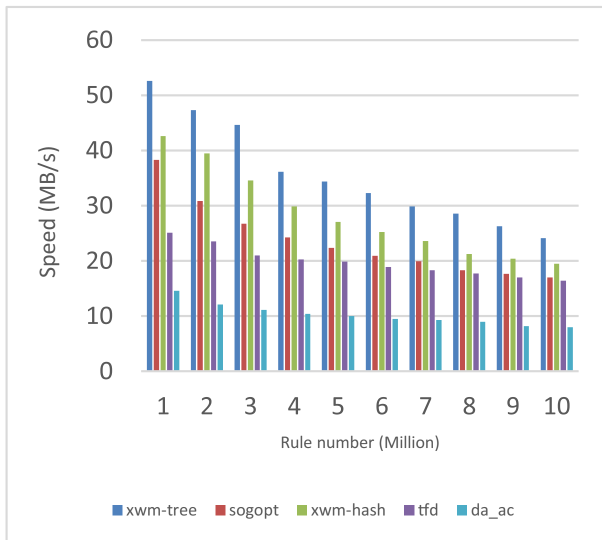

Fig. 5 Comparison of speeds of different algorithms when the shortest pattern string length is 10

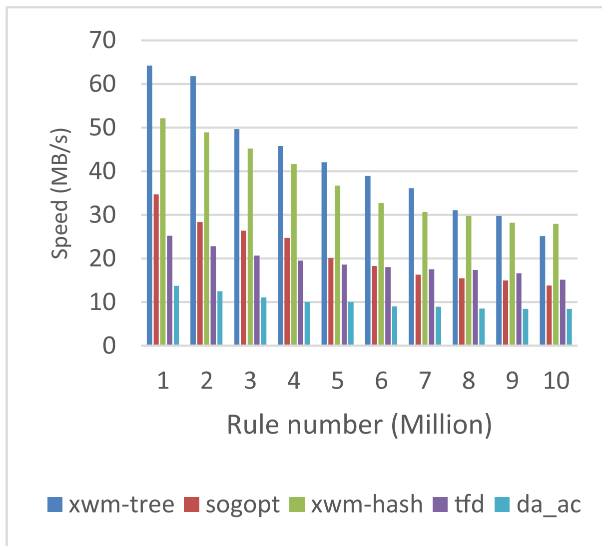

Fig. 6 Comparison of speeds of different algorithms when the shortest pattern string length is 12

XWM-Hash, and SOGOPT had roughly equivalent matching speeds when using 10 million pattern strings. All of them had higher matching speeds than the double-array AC algorithm and TFD algorithm. The advantages of our algorithm become clearer when the length of the shortest pattern string is longer. Both XWM-Tree and XWM-Hash had matching speeds about twice that of other algorithms with the longer shortest pattern string lengths.

However, Performance of the SOGOPT algorithm gradually decreased as the length of the shortest pattern string increased leading to the decreasing number of groups of packet reduction and the increasing number of verifications. The AC and TFD algorithms were less

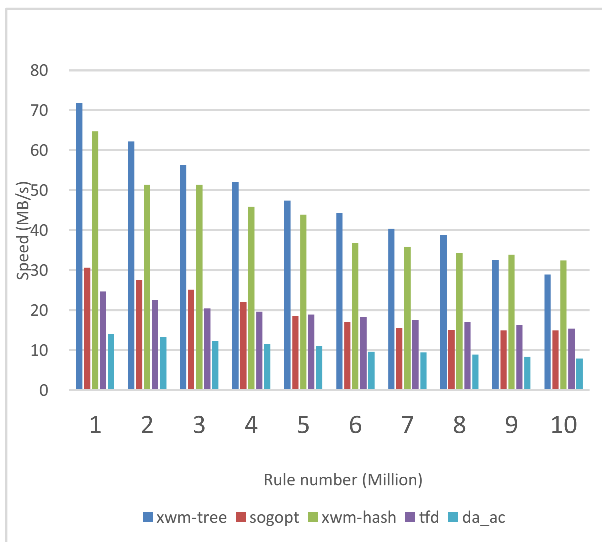

Fig. 7 Comparison of speeds of different algorithms when the shortest pattern string length is 14

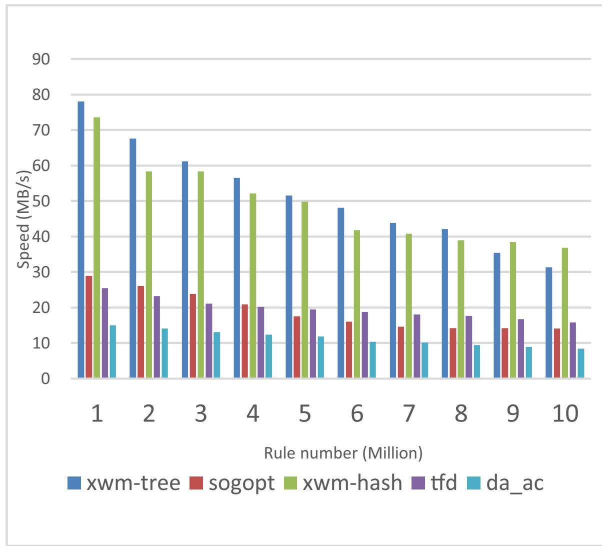

Fig. 8 Comparison of speeds of different algorithms when the shortest pattern string length is 16

affected by the length of the shortest pattern strings and the number of pattern strings due to the trie data structure, showing a relatively stable matching performance. The matching speed of our XWM-Hash algorithm is slightly slower than XWM-Tree algorithm.

Since the unit of double-array AC and TFD is too large to identify the proposed method and the result of SOGPOT, we show the results of XWM-Tree, XWM-Hash, and SOGPOT only in Figs. 9, 10, 11, 12, 13 and 14. The speed and memory of the double-array AC and TFD algorithms in different shortest pattern string length (SPSL) are shown in Table 3, respectively.

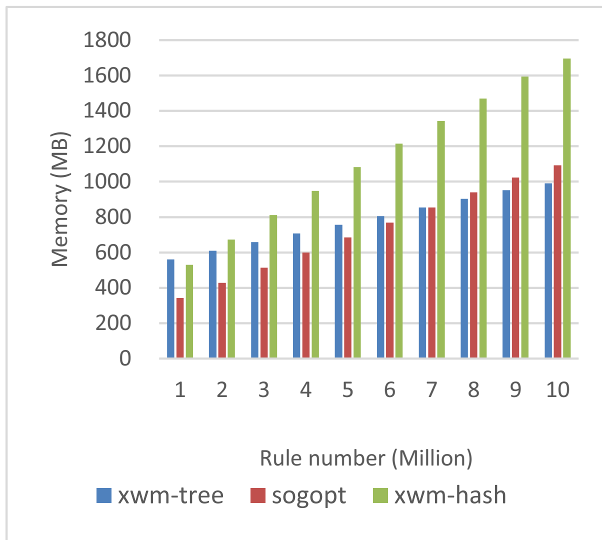

Fig. 9 Comparison of memory of different algorithms when the shortest pattern string length is 6

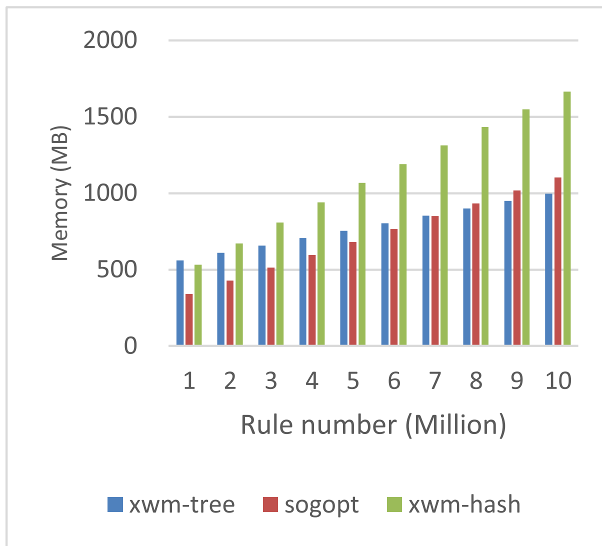

Fig. 10 Comparison of memory of different algorithms when the shortest pattern string length is 8

The experimental results show that XWM-Tree and XWM-Hash algorithm used less memory thanks to the two compressed hash tables. When the number of pattern strings was between 1 million and 7 million, our algorithm used slightly more memory than SOGOPT. However, when the number of pattern strings reached 10 million, both XWM and SOGOPT used considerable memory, and both used significantly less than the TFD and double-array AC algorithms. Since the XWM-Hash algorithm used a hash table to organize the conflict pattern string, the memory consumption was greater than the XWM-Tree algorithm. Even so, XWM-

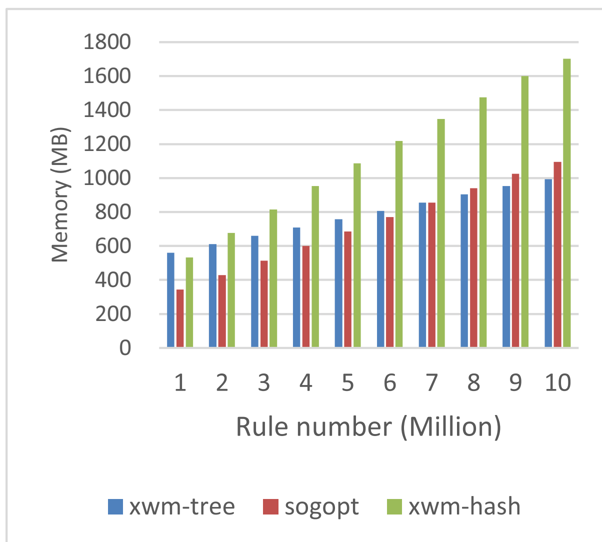

Fig. 11 Comparison of memory of different algorithms when the shortest pattern string length is 10

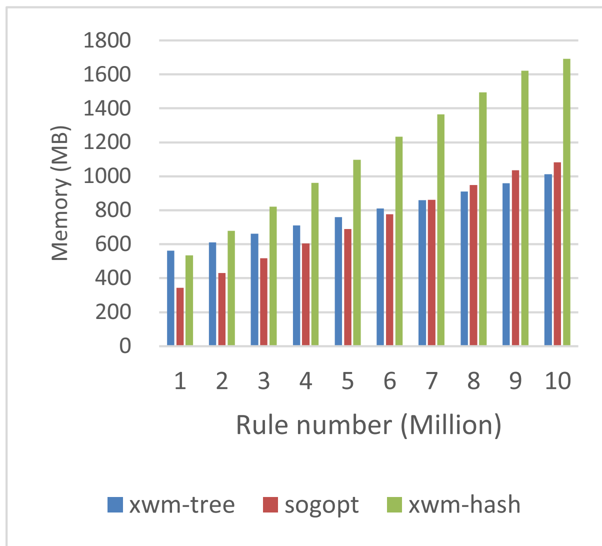

Fig. 12 Comparison of memory of different algorithms when the shortest pattern string length is 12

Hash consumed less than 2 GB of memory when handling 10 million pattern strings, which was much lower than the nearly 8 GB used by double-array AC and TFD.

It can be seen in Figs. 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 and 14 that the matching performance of the XWM-Tree algorithm is higher than the XWM-Hash algorithm and the memory usage of XWM-Tree algorithm is lower than the XWM-Hash algorithm. This is because, in the XWM-Hash algorithm, some pattern strings are often hashed to the same bucket, which may result in the need to compare the pattern strings one by one when matching. Comparatively, in the XWM-Tree algorithm, after processing the pattern string based on the window selection technique, there are very

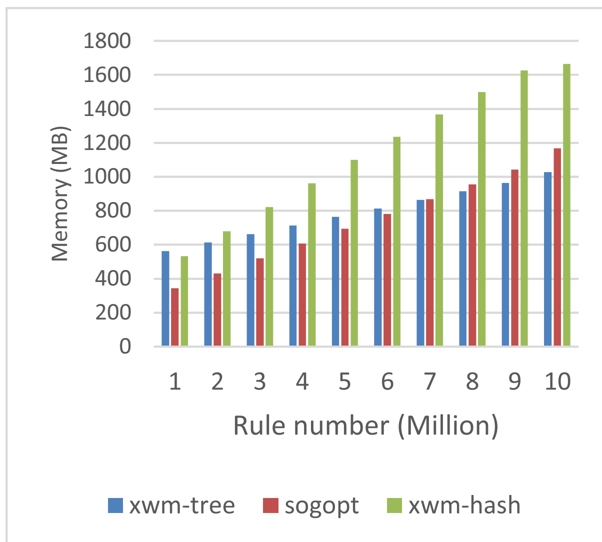

Fig. 13 Comparison of memory of different algorithms when the shortest pattern string length is 14

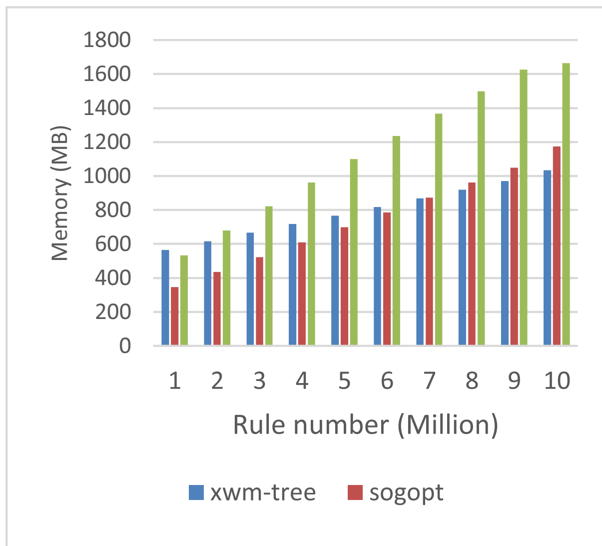

Fig. 14 Comparison of memory of different algorithms when the shortest pattern string length is 16

few matching windows with exactly the same prefix, so the matching performance of the XWM-Tree algorithm is higher than the XWM-Hash algorithm. While in memory usage, the XWM-Tree algorithm consumes less memory than XWM-Hash because the tree-based organization structure in the associative container is more compact and less wasteful than the hash-based organization.

5 Conclusion

In this paper, we proposed XWM algorithm to match large numbers of URL rules. It reduces hash collisions and the number of precise comparisons by adapting to the specific characteristics of URLs. Experimental results with real data show that the matching speed of XWM is doubled faster than traditional algorithms, which makes it more suitable and preferable for

Table 3 Memory usage (MB) of both TDF and AC algorithms

SPSL	algrithm	Rule number(million)									
		1	2	3	4	5	6	7	8	9	10
6	tfd	565.7	1172.5	1943.9	2805.4	3698.9	4591.8	5456.3	6281.8	7054.3	7645.7
	da_ac	784.0	1544.8	2296.5	3041.8	3782.3	4519.3	5252.1	5983.3	6712.1	7397.7
8	tfd	571.2	1173.8	1936.9	2784.3	3665.9	4545.6	5399.2	6214.7	6982.9	7695.1
	da_ac	790.8	1558.3	2316.6	3068.2	3815.2	4558.4	5297.3	6034.8	6769.9	7502.3
10	tfd	566.7	1174.5	1947.2	2810.3	3705.3	4599.7	5465.7	6292.6	7066.4	7658.8
	da_ac	794.7	1565.9	2327.9	3083.4	3834.0	4581.2	5324.0	6065.2	6804.0	7499.0
12	tfd	565.0	1178.8	1965.8	2845.6	3763.3	4674.5	5558.8	6395.3	7178.3	7885.8
	da_ac	812.1	1600.4	2378.8	3151.1	3918.7	4681.9	5441.5	6199.3	6954.1	7770.3
14	tfd	567.6	1175.7	1950.0	2813.4	3711.5	4606.6	5474.6	6300.9	7175.9	7946.6
	da_ac	830.3	1636.9	2433.4	3223.4	4008.6	4789.8	5567.4	6343.0	7115.2	7938.9
16	tfd	568.8	1178.1	1953.9	2819.1	3719.0	4615.9	5485.7	6313.6	7190.4	7962.7
	da_ac	845.1	1666.2	2477.0	3281.2	4080.4	4875.6	5667.2	6456.7	7242.7	8081.2

large-scale application environments. The XWM algorithm's performance depends on hash functions to construct shift table and hash table, and organize associative containers. It is suitable for large-scale rule sets with longer lengths of the shortest pattern string, especially when the length of shortest pattern string is 10 or more. The future work will include improving the performance of algorithm to adapt general patterns.

Acknowledgements This study is supported by National key research and development program (2016YFB0801205).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Aho AV, Corasick MJ (1975) Efficient string matching: an aid to bibliographic search[J]. Commun ACM 18(6):333–340
2. Allauzen C, Crochemore M, Raffinot M (1999) Factor oracle: A new structure for pattern matching[C]. International Conference on Current Trends in Theory and Practice of Computer Science. Springer Berlin Heidelberg 295–310
3. Du X, Xiao Y, Guizani M et al (2007) An effective key management scheme for heterogeneous sensor networks[J]. Ad Hoc Netw 5(1):24–34
4. Du X, Guizani M, Xiao Y, Chen HH (2009) Transactions papers, "a routing-driven elliptic curve cryptography based key management scheme for heterogeneous sensor networks,". IEEE Trans Wirel Commun 8(3):1223–1229
5. Kalnoor G, Agarkhed J (2016) Pattern matching intrusion detection technique for Wireless Sensor Networks[C]. Int Conf Adv Electr IEEE
6. Kalnoor G, Agarkhed J (2018) Detection of intruder using KMP pattern matching technique in wireless sensor networks[J]. Procedia Comput Sci 125:187–193
7. Liu YB, Shao Y, Wang Y, Liu QY, Guo L (2014) A multiple string matching algorithm for large-scale URL filtering. Chin J Comput 37:1159–1169
8. Navarro G, Raffinot M (1998) A bit-parallel approach to suffix automata: fast extended string matching[C]. Combinatorial Pattern Matching. Springer Berlin/Heidelberg 14–33
9. Navarro G, Raffinot M (2002) Flexible pattern matching in strings. Practical on-line search algorithms or texts and biological sequences. Cambridge 1-2
10. Qiu J, Chai Y, Liu Y, Gu ZQ, Li S, Tian Z (2018) Automatic non-taxonomic relation extraction from big data in Smart City[J]. IEEE Access 6:74854–74864
11. Salmela L, Tarhio J, Kytöjoki J (2007) Multi-pattern string matching with q-grams[J]. J Exp Algorithmics (JEA) 11:1.1
12. Tan Q, Gao Y, Shi J, Wang X, Fang B, Tian ZH (2018) Towards a comprehensive insight into the eclipse attacks of Tor hidden services. IEEE Internet Things J. https://doi.org/10.1109/JIOT.2018.2846624
13. Tian Z, Su S, Shi W, Yu X, Du X, Guizani M (2019) A data-driven model for future internet route decision modeling[J]. Futur Gener Comput Syst 95:212–220
14. Tian Z, Li M, Qiu M, Sun Y, Su S (2019) Block-DES: a secure digital evidence system using Blockchain. Inf Sci 491:151–165. https://doi.org/10.1016/j.ins.2019.04.011
15. Tian Z, Shi W, Wang Y, Zhu C, Du X, Su S, Sun Y, Guizani N (2019) Real time lateral movement detection based on evidence reasoning network for edge computing environment. IEEE Trans Ind Inf. https://doi.org/10.1109/TII.2019.2907754
16. Tian Z, Gao X, Su S, Qiu J, Du X, Guizani M Evaluating reputation management schemes of internet of vehicles based on evolutionary game theory. IEEE Trans Veh Technol. https://doi.org/10.1109/TVT.2019.2910217
17. Wang Y, Sun M, Wang K et al (2016) Quality of experience estimation with layered mapping for hypertext transfer protocol video streaming over wireless networks[J]. Int J Commun Syst 29(14):2084–2099
18. Wu S, Manber U (1994) A fast algorithm for multi-pattern searching [J]
19. Xiong G, He HM, Yu J et al (2015) HybridFA: A memory reduction technique for the AC automata based on statistics[J]. J Commun

20. Xu DL (2015) Research on high-performance online pattern matching algorithm[D]. Harbin Institute of Technology
21. Yu X, Tian Z, Qiu J, Jiang F (2018) A data leakage prevention method based on the reduction of confidential and context terms for smart Mobile devices[J]. Wirel Commun Mob Comput 2:1–11
22. Yuan Z, Yang B, Ren X et al (2013) TFD: A multi-pattern matching algorithm for large-scale URL filtering[C]. Computing, Networking and Communications (ICNC), 2013 International Conference on IEEE 359–363
23. Zhang P, Liu YB, Yu J et al (2015) HashTrie: a space-efficient multiple string matching algorithm[J]. J Commun

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Shuzhuang Zhang , Doctor of Information Security, Lecturer. Graduated from the Harbin Institute of Technology in 2011. Worked in Beijing University of Posts and Telecommunications. His research interests include Network Security and Information Content Security.

Yanbin Sun received the B.S., M.S. and Ph.D degree in Computer Science from Harbin Institute of Technology (HIT), Harbin, China. He is currently an assistant professor in Guangzhou University, China. His research interests include network security, future networking and scalable routing.

Fanzhi Meng , Institute of Computer Application, China Academy of Engineer Physics, Mianyang, 621900, China

Yunsheng Fu , Institute of Computer Application, China Academy of Engineer Physics, Mianyang, 621900, China

Bowei Jia , Master of Computer Science and Technology, His research interest is Network Security.

Zhigang Wu , Doctor of Computer Architecture, Associate Professor. Graduated from the Harbin Institute of Technology in 1996. Worked in Beijing University of Posts and Telecommunications. His research interests include Network Security and Information Security.