A Data-Deduplication-Based Matching Mechanism for URL Filtering

Yuhai Lu, Yanbing Liu*, Chunyan Zhang, Jianlong Tan
Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
University of Chinese Academy of Sciences, Beijing, China

***Abstract*—URL filtering plays an important role in various network security applications. URL filtering usually requires high matching performance, but the performance of the classical multiple string matching algorithms have been difficult to be significantly improved. In this article, we found that the online URLs to be filtered contain a large number of duplicate URLs. According to this observation, we propose a novel deduplication-based matching mechanism (DBM) for URL filtering. The DBM caches information of the duplicate URLs in a hash table to avoid duplicate URLs being repeatedly scanned by URL filtering system. The DBM can be used in conjunction with any multiple string matching algorithms. Experimental results show that when a multiple string matching algorithm used in conjunction with the DBM, the matching speed of the URL filtering system can be increased by 9%-68%. So DBM can significantly accelerate the speed of URL filtering system. Besides increasing speed of URL filtering system, DBM is a mechanism independent of the specific matching algorithm and can be easily used in other field.**

***Index Terms*—multiple string matching; URL filtering; data deduplication; network security**

I. Introduction

URL filtering has been widely used in a variety of network security systems, such as Firewall and IDS/IPS. Filtering out those malicious URLs effectively protect people from attacked by the harmful information. All the time, multiple string matching algorithm has been the most direct and effective method to implement URL filtering. In this article, multiple string matching used in URL filtering refers to sub-string matching, that means the URL rule is the sub-string of the URL to be scanned, but not a whole URL. With the widespread use and rapid development of modern internet, the amount of website grows fast, together with malicious websites. This brings two critical challenges to URL filtering system. On the one hand is that the number of malicious URL rules has been enormous, even reaching several millions, the matching speed of multiple string matching algorithm decreases rapidly with the increase of rule scale. On the other hand the development of network technology accelerates the growth of network bandwidth, high network bandwidth requires faster matching speed of the URL filtering system.

So far, many different algorithms have been applied to the URL filtering system, such as those classical algorithms and some improved algorithms. Looking back upon these existing works, we can find that all of them have devoted to speeding up the URL filtering system by improving the multiple matching algorithms. For the URL filtering, the total number of URL existing in the internet is limited, so the online URLs to be filtered must contain a large number of duplicate URLs. We captured 29 GB (about 300 million) size of URLs from the backbone network and counted their repetition rate. The result is show at Figure 1. The repetition rate is about 64.5%-80% with the URL number increases from 2,000,000 to 300,000,000, and we can find that the repetition rate increases as the number of URLs to be filtered increases.

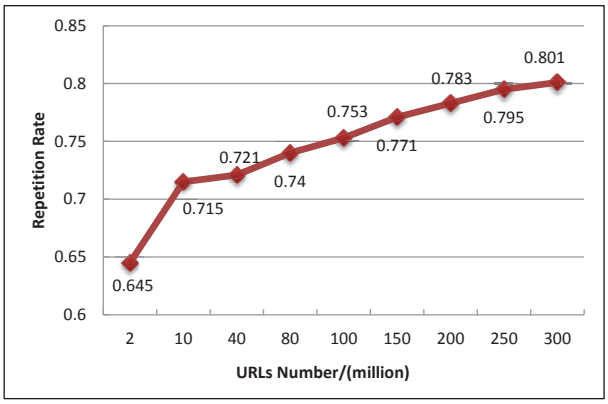

Fig. 1. Repetition rate of online URLs.

Generally speaking, the speed of the data deduplication is faster than the speed of the multiple string matching. In view of this, we propose a novel deduplication-based matching mechanism (DBM) for URL filtering. The DBM is a flexible matching mechanism above specific multiple string matching algorithms, which can be used in conjunction with any multiple string matching algorithm. We compared the DBM with different alone multiple string matching algorithm, the matching speed of the URL filtering system can be increased by 9%-68%.

The rest of this article is organized as follows. Section II reviews the related work by analyzing representative works in URL filtering. Section III introduces the key ideas and workflow of DBM. Experimental results and comparison with some existing algorithms are detailed and analyzed in Section IV. Section V concludes the article.

II. Related Work

Along with the widely used of URL filtering, many different multiple string matching algorithms have been used in URL

*Corresponding author: Email: liuyanbing@iie.ac.cn

978-1-5386-3180-5/18/$31.00 ©2018 IEEE

filtering system. According to the characteristics of the multiple string matching algorithms, we divide the existing classical algorithms into three categories: automata based matching algorithm, hash based matching algorithm and bit parallel based matching algorithm. (1) Automata based matching algorithms compile the rules into a whole automaton, and then scan the data by the automata. AC[1] and SBOM[2] are typical automata based matching algorithms. The performance of these algorithms is stable on different types of data sets and has linear scan speed in theory. But disadvantage is that automata need to consume so huge storage space that it cannot adapt to large-scale string matching. (2) Hash based matching algorithms calculate the hash value of the rules and store the hash values in the hash table, KR[3] and WM[4] are typical hash based matching algorithms. The advantage of hash based matching algorithms is that they only need small storage space and get fast matching speed on random data with large character sets. But for data with small character sets or for rules with short length, the matching speed of these algorithms drops rapidly. (3) Bit parallel based matching algorithms use bit vector to simulate the automata, the operation of the bit vector is used to represent the state transition of automata. Shift-AND/Shift-OR[5] and BNDM[6] are typical bit parallel based matching algorithms. These algorithms have the advantages of small storage space consuming and fast matching speed. But bit parallel based matching algorithms are only suitable for small-scale (only dozens of pattern strings) string matching.

In recent years, researchers have proposed some matching algorithms specifically for URL filtering. Algorithm in [7] combines a URL compression algorithm with a multiple string matching based (Wu-Manber-like) matching algorithm. They adopt multi-phase hash and dynamic-cut heuristics strategies. From this method, the proposed URL lookup engine can achieve high URL lookup performance and efficient memory utilization for storing the ever-increasing URL blacklist with the ability of prefix matching. But this method only support prefix matching of URL, and does not support the substring matching, thus limits its application in some degree. Algorithm in [8] generates two filtering models based on URL datasets to use before existing processing methods. They use lexical features and descriptive features of URL string, and then combine the filtering results. On-line learning algorithms are applied to deal with large-scale data sets and fit the very short lifetime characteristics of malicious URLs. According to their filter, it can significantly reduce the volume of URL queries on which further analysis needs to be performed, saving both computing time and bandwidth. In [9] it proposes a TFD algorithm to handle large scale (over ten million URLs) URL filtering. It employs Four-byte Block, Two-phase hash, Finite state machine and Skip after Exact Matching to eliminate the performance bottleneck of blacklist filter. It absorbs the idea of Double-Array algorithm by using Double-Array Storage to tackle the memory explosion brought by FSM. Experimental results show that TFD optimizes the WM algorithm effectively, and achieves a hundreds of times acceleration when dealing with pattern sets of 10 million URLs.

We can conclude that all of the existing matching algorithms improved the URL filtering system by optimizing the performance of the matching algorithms, but all of them have not noticed the important statistical feature of the on line URLs that online URLs to be filtered contains a large amount of duplicate URLs. According to the observed feature, in this article, we propose a novel data deduplication mechanism for URL filtering.

III. DBM: A DATA-DEDUPLICATION-BASED MATCHING MECHANISM

A. Data Deduplication Mechanism

As previously described, a URL filtering system faces a large number of duplicate URLs, the repetition rate has been showed at Figure 1. Scanning every URL on line by using string matching engine is a waste of computing resources and affects the scanning speed. For those duplicate URLs, we can use data deduplication mechanism instead of scanning them by using string matching engine to accelerate the URL filtering.

In this article, we propose a novel Data-Deduplication-Based Matching Mechanism (DBM). DBM is independent of specific matching algorithm. It can be used in conjunction with multiple string matching algorithms or the regular expression matching algorithms. Sub-figure b of Figure 2 shows the architecture of DBM for URL filtering, and traditional architecture of URL filtering is showed at sub-figure *a* of Figure 2. From sub-figure *b* of Figure 2, we can know that DBM uses an extra cache pool to store the information of those distinct URLs. Algorithm 1 describes the procedure of scanning a new URL by DBM. When a new URL is going to be scanned, firstly it will be compared with those URLs stored in the URL cache pool. If it has showed up before, DBM will return the matching information that cached in URL cache pool directly. If not, the DBM will use multiple string matching engine to scan this URL, then return the matching result and cache the information about this URL into the URL cache pool, like its hash value and matching result.

Furthermore, we can analyze the principles and feasibility for data deduplication mechanism. In URL dataset, let p be the percentage of distinct URLs, then the repetition percentage of this URL dataset is *1-p*. During the phase of URL scanning, suppose T_1 as the time of each data deduplication, T_2 as the time of each string matching, and T_3 as the time of each inserting URL into the URL cache pool. Thus, the expected URL scanning time is T

$$\begin{aligned} T &= (1-p)T_1 + p(T_1 + T_2 + T_1) \\ &= T_1 + p(T_2 + T_3) \end{aligned} \tag{1}$$

The speed of data deduplication mechanism is usually much faster than the speed of multiple string matching algorithms, so T_1 and T_3 are very small, and can make $T < T_2$. That means, compared with traditional URL filtering method, data deduplication mechanism can accelerate the speed of the URL filtering system.

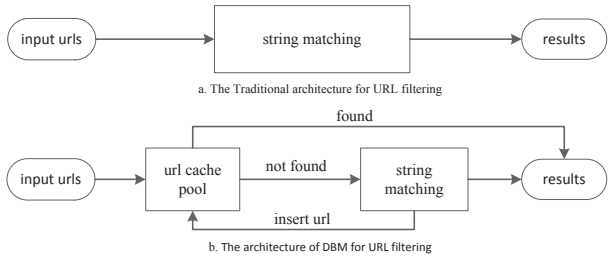

Fig. 2. Architecture of DBM and traditional method.

In order to implement the goal of data deduplication, we can choose a variety of different methods. The easiest method is to store each original URL in URL cache pool, but it is also the worst method. Because, when we check a new input URL if it has been stored in the URL cache pool, we have to use the new input URL to compare with all of the URLs stored in the URL cache pool one by one, this comparing is slower than the string matching and cannot accelerate the URL filtering. Besides, it will consume a large amount of main memory to store the distinct original URLs.

So in this article, we choose the hash method to realize data deduplication. Each URL′s information, including two 64-bit hash values and the match information, are stored in a hash node, and those URLs with the same hash table address will be organized into a link list.

Algorithm 1 Scanning a new URL by DBM

```
1: procedure SCANURL(URL[1...n])
2:     (h0, h1)=DPHash(URL[1...n])
3:     i=h0 mod (N)
4:     hash_node=hash_table[i]
5:     while hash_node.next!=NULL do
6:         if ((h0, h1)==(hash_node.h0, hash_node.h1)) then
7:             return hash_node.m
8:         else
9:             hash_node=hash_node.next
10:        end if
11:    end while
12:    m=StringMatching(URL[1...n])
13:    new_hash_node=NewNode(h0, h1, m)
14:    hash_table[i]=new_hash_node
15:    new_hash_node.next=hash_node
16:    return m
17: end procedure
```

B. Hash Algorithm Selection Strategy

There are many classic hash algorithms that have been applied in various fields. So we don′t need to design a completely new hash algorithm, but choosing a classic hash algorithm as the basic hash algorithm for the data deduplication mechanism. In order to select an efficient hash algorithm, the algorithm′s computational speed and collision rate are the two main considerations.

When designing the hash algorithm for data deduplication mechanism, we referred to the linear congruential generator(LCG) introduced in [14] and uses polynomial hash (PLHash) algorithm as the basics.

LCG is an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation. The formula of generator is $x_{n+1}=(a*x_n+c) mod (m)$, where x is the sequence of pseudo-randomized values, a and c are two integer constants, m is the range of the pseudo-random number to be generated.

The PLHash algorithm is the basic hash algorithm used in our DBM. The formula of the algorithm is $h=a*h+data_i$, where a is a constant, $data_i$ is the i-th character of the input data, h is the hash value.

Referring to the linear congruential generator and the PLHash algorithm, we designed a Double Poly Hash (DPHash) algorithm for data deduplication mechanism. The DPHash is recursively defined as:

$$h_0 = a_0 * h_0 + url_i \tag{2}$$

$$h_1 = a_1 * h_1 + url_i \tag{3}$$

For a_0 and a_1 are two constants, url_i is the i-th character of the input URL, h_0 and h_1 are two 64-bit integral hash values. We use the two 64-bit integers to uniquely characterize a URL.

The detail calculating procedure of DPHash algorithm is described at Algorithm 2. Two 64-bit integers can be equivalent to a 128-bit integer, the same as the result of the MD5 algorithm. So we can infer that the collision rate of the DPHash can be very low, which can be similar to the collision rate of MD5 algorithm. Furthermore, we can theoretically analyze the collision rate of the DPHash algorithm. According to birthday attack theory introduced in [18], with a birthday attack, it is possible to find a collision of a hash function in $\sqrt{2^n}=2^{n/2}$, with 2^n is the range of hash value. As for the DPHash algorithm, it is possible to find a collision for 2^{64} times different inputs, this is far larger than the total amount of URL in reality, so the collision rate will be extremely low.

Algorithm 2 Calculating a URL′s hash value by DPHash

```
1: procedure DPHASH(URL[1...n])
2:     uint64 a0=6364136223846793005
3:     uint64 a1=2862933555777941757
4:     uint64 h0=1442695040888963407
5:     uint64 h1=3037000493
6:     for i=1 to n do
7:         h0=a0*h0+URL[i]
8:         h1=a1*h1+URL[i]
9:     end for
10:    return (h0, h1)
11: end procedure
```

In order to gain a good hash effect, we arranged initial value for h_0 and h_1. The pairs (h_0, a_0) and (h_1, a_1) are just like the pair (a, c) used in pseudo-randomized number

TABLE I
PERFORMANCE COMPARE FOR DIFFERENT HASH ALGORITHMS.

hash name	hash value range	collision rate	speed(MB/s)
CRC	2^{32}	0.007	340.875
RSHash	2^{32}	0.007	518.281
JSHash	2^{32}	0.016	493.481
APHash	2^{32}	0.013	428.344
BPHash	2^{32}	0.988	652.157
SDBHash	2^{32}	0.007	502.861
DJBHash	2^{32}	0.007	582.818
ELFHash	2^{32}	0.743	399.909
FNVHash	2^{32}	0.007	507.686
DEKHash	2^{32}	0.008	650.066
BKDRHash	2^{32}	0.007	516.521
PJWHash	2^{32}	0.743	395.491
DPHash	$\mathbf{2^{128}}$	**0.000**	**533.037**
MD5	2^{32}	0.000	286.605

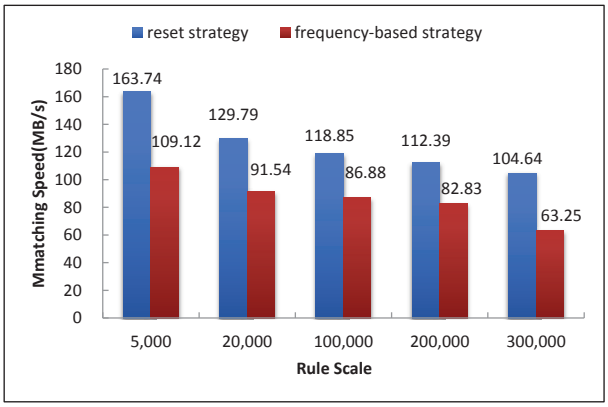

Fig. 3. Performance compare for different clear strategy.

generator in [14] . The initial value of the h_0 is a 64-bit integer 6364136223846793005, value of constant a_0 is 64-bit integer 6364136223846793005, these two integers are two parameters of pseudo-randomized number generator used in MMIX[15] . The initial value of the h_1 is 64-bit integer 3037000493, value of constant a_1 is 64-bit integer 2862933555777941757, these two integers are two parameters of pseudo-randomized number generator used in Computational Nuclear Physics Group[16].

In order to evaluate the performance of the hash algorithms, we evaluated the computing speed and the collision rate of different hash algorithms. The result is showed at Table I. The test data is about 63 million (size is about 12 GB) real online URLs with no repeat, which are captured from the backbone network. In our experiment, we selected 14 kinds of common hash algorithms, where DPHash is used in our proposed, MD5[11] and CRC[12] are two classical algorithms widely used in a variety of areas, and the others are some common hash algorithms introduced in [17].

From Table I, we can see that the collision rate of MD5 and DPHash is 0, but the DPHash′s speed is 1.9 times faster than MD5. There are other algorithms which are faster than the DPHash, but their collision rate are also higher than DPHash, especially the fastest BPHash has a collision rate up to 0.988. In DBM, a collision means that two different URLs have the same hash value, that maybe lead to incorrect matching results. So we need an extremely low collision rate to ensure the correctness of the matching results. Meanwhile, the matching speed must be fast enough, so from Table I, we can know that the DPHash makes the best balance between matching speed and collision rate, so we select DPHash for our proposed.

C. Cache Pool Clear Strategy

As we all know that main memory space of a computer is limited, and at the start of the DBM, we need to set the size of the hash table in advance, which means the number of URLs that can be cached in URL cache pool is limited. So with the increasing of URLs to be filtered, cache capacity cannot be always enough. Especially during the practical use, the URL amount of a URL filtering system receives can arrive at a million or more per hour on average. So when hash table is full, we need a clear strategy to clear some URLs have been stored in the URL cache pool.

In this article, we implement two different clear strategies, one is reset clear strategy, and the other is frequency-based clear strategy. The first reset clear strategy will reset the hash table when it is full. The advantage of this strategy is that it can be simply implemented, but once the hash table is reset, it wills increase the times of invoking multiple string matching engine. The second frequency-based clear strategy takes into account the frequency of URL access. From previous analysis and statistical feature, we can know that the probability of URLs to be accessed in real world has a significant difference, that is to say the frequency of URL access is very different. Therefore the frequency-based clear strategy will clear a certain percentage (such as 20%) of the lowest access frequency of URLs when hash table is needed to be cleared. The frequency-based clear strategy is more reasonable and guarantees that the hash table will not be emptied. But it need to sort the URLs according to the frequency and update the hash table after clearing, these operations increase the complexity of the method and slow down the matching speed.

We compare the matching performance for the reset strategy and the frequency-based strategy. The string matching algorithm used in this experiment is AC, and we test for different rule scale. The result is showed at Figure 3.

From Figure 3 we find that the performance of reset strategy is better than the frequency-based strategy. So in our proposed, we select the reset strategy for the DBM.

IV. EXPERIMENT AND EVALUATION

Until now, many different multiple string matching algorithms have been proposed. We have mentioned that the proposed DBM can be used in conjunction with any multiple string matching algorithm, so we will not evaluate the DBM for all of the existing multiple string matching algorithms. In our experiment, we use DBM in conjunction with two different multiple string matching algorithms, AC[1], and MMBLG (Matching method based on length grouping), to evaluate performance of the DBM. MMBLG is a combination algorithm,

which can divide the rules into two groups according to the rule length, those rules with length less than 6 will be preprocessed by AC[1] algorithm, and other rules will be preprocessed by KR[3] algorithm.

When DBM used in conjunction with a specific multiple string matching algorithms, it will not increase the time of the preprocessing procedure, so we will not evaluate the preprocessing time in our experiment. In our experiment, we mainly analyze the matching speed improving and influence of cache pool size.

A. Experimental Setup

The hardware and software environment of the experiment is as follows: CPU is Intel Xeon E5-2667(3.20GHz), main memory size is 125 GB, and operating system is CentOS 7.2 (64 bit). The algorithms are written in C++ and executed by single thread.

We have mentioned that we captured 29 GB (about 300 million) size of URL from the backbone network, it is used as the input URLs to be scanned. The URL rule sets are randomly generated, the length range of URLs rule is 4-256, and the character set is all of the 29 GB URLs characters.

B. Matching Speed

For a clearer analysis of the speed of growth, we give the Speedup Ratio as the level of speed increase. Speedup Ratio is defined as following: Given two different kinds of algorithms A and B, let MS_A and MS_B be the matching speed of algorithm A and algorithm B separately. So Speedup Ratio of algorithm A for algorithm B is:

$$Speedup\ Ratio = (MS\_A\text{-}MS\_B)/MS\_B \tag{4}$$

During the matching speed experiment, the size of hash table for the cache pool is fixed to 2^{24}, which can be dynamically set according to the available main memory size of the machine.

The specific performance of DBM in conjunction with AC and MMBLG for different rule scale is respectively showed at Figure 4 and Figure 6. The speedup ratio for AC and MMBLG with different rule scale is respectively showed at Figure 5 and Figure 7. The largest rule scale for AC is 300,000, which is because the memory consumption is too large when scale is over 300,000. The experimental results of AC and MMBLG both show that DBM can effectively accelerate the matching speed of URL filtering system, the matching speed of the URL filtering system increases by 9%-68%. Both Figure 5 and Figure 7 show that the speedup ratio increases as the rule scale increases. For the MMBLG algorithm, when rule scale is 5,000, speedup ratio is 0.299, but when rule scale is 5,000,000, speedup ratio is up to 0.628. This is because the matching speed is fast enough for the small rule scale, so the speed acceleration of DBM is not so obvious. But with the increase of rule scale, the speed of matching algorithm is declining rapidly, but the speed of data deduplication will not decrease significantly, so the acceleration effect of DBM is more obvious. That means DBM′s acceleration can be more effective for large-scale URL filtering.

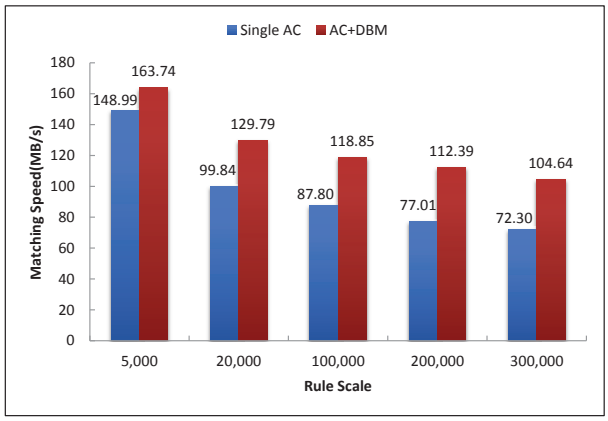

Fig. 4. Performance of DBM in conjunction with AC for different rule scale.

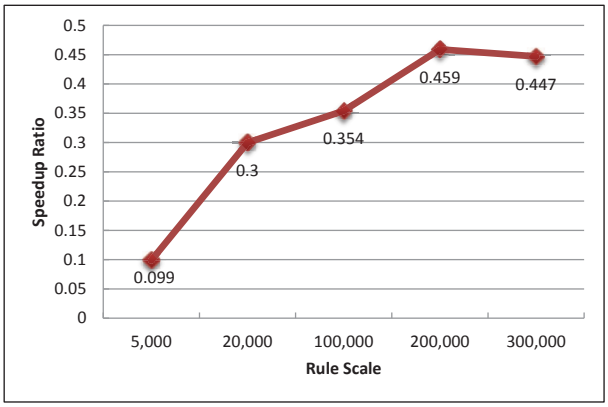

Fig. 5. Speedup ratio for AC with different rule scale.

C. Influence of Cache Pool Size

As mentioned earlier, when the cache pool is full, some space need to be cleared for the new input URLs. When clearing the cache pool, no matter reset clear strategy or the frequency-based clear strategy, both two maybe reduce the matching speed. So we compared the matching speed with

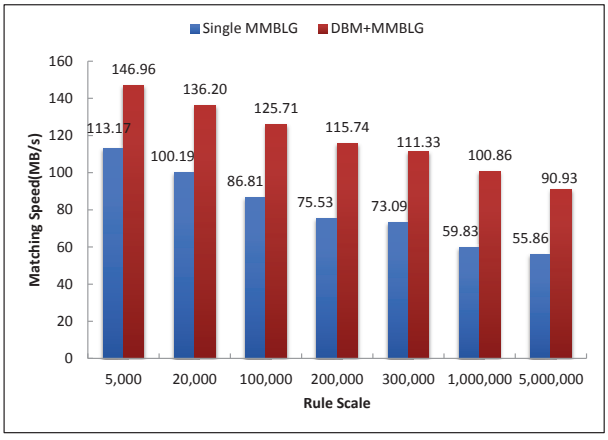

Fig. 6. Performance of DBM in conjunction with MMBLG for different rule scale.

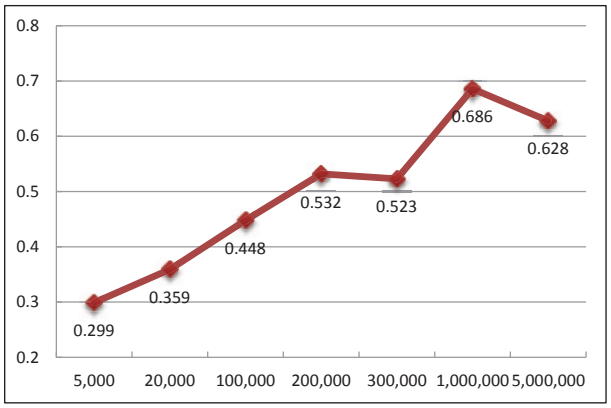

Fig. 7. Speedup ratio for MMBLG with different rule scale.

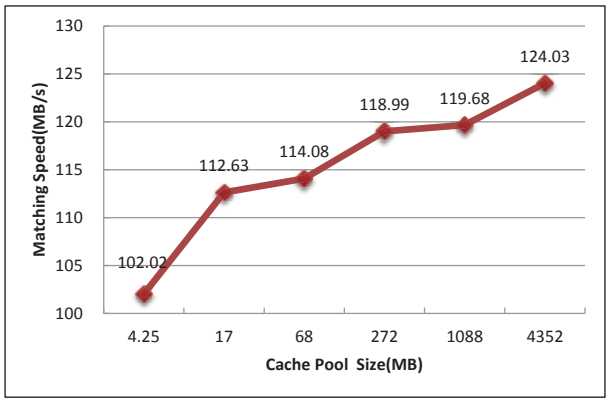

Fig. 8. Matching speed for different cache pool size.

different cache pool size, result is show at Figure 8. The matching algorithm used in this experiment is AC, and the rule scale is 100,000, the minimal hash table size is 2^{16}, and increases to 2^{26} with 4 times increase step.

From Figure 8, we can see that the matching speed increases as the cache pool size increases. As cache pool size increases from 4.25 MB to 4352 MB, the matching speed increases by 21%. The experimental result shows that the cache pool size could influence the matching speed, that because the larger the cache pool size is, the lower the clearing frequency is, in contrast, the smaller the hash table size is, the higher the clearing frequency is. In real URL filtering system, we can dynamically set the cache pool size according to the available main memory size of the machine. If main memory size is large enough, we should set the cache pool as large as possible.

V. Conclusion

In this article, we proposed a Data-Deduplication-Based Matching Mechanism (DBM) for URL filtering. It can effectively accelerate matching speed by using hash technique to deduplicate the original URL datasets. Experimental result indicates that our approach can arrive at a 68% speedup ratio at most compared with the classical algorithm MMBLG without data deduplication. This acceleration is much more obvious when concern with large-scale (with several million URL rules) in real-time scenarios. In addition, DBM is a mechanism independent of the specific matching algorithm, it can be easily applied to other fields with similar scenes.

The proposed DBM also has two shortcomings. One is that we used a reset strategy to handle cache problem when cache pool is full, it is not the best clear strategy. Therefore, our future work will keep studying other effective methods to refresh the cache pool, such as improving the frequency-based strategy. The other is that DBM maybe return false matching result in theory. Although collision rate of DPHash used in DBM is extremely low, but just like MD5 algorithm, the possibility of hash collision still exists, and the hash collision maybe lead to false matching result.

VI. Acknowledgment

This work is partially supported by The National Key Research and Development Program of China (No.2016YFB0800303 and No.2016YFB0801300), and the Fundamental theory and cutting edge technology Research Program of Institute of Information Engineering, CAS(No. Y7Z0351101).

References

[1] A. V. Aho and M. J. Corasick, Efficient string matching: an aid to bibliographic search, *Communications of the ACM*, 18(6):333-340, 1975
[2] Allauzen C, Crochemore M, Raffinot M, Factor Oracle: A New Structure for Pattern Matching, *Theory and Practice of Informatics*, 1999, pp.295-310.
[3] Richard M. Karp, Michael O. Rabi, Efficient Randomized Pattern-Matching Algorithms, *IBM Journal of Research and Development*, 31(2), 1987, pp.249-260
[4] Wu S, Manber U, A fast algorithm for multi-pattern searching, *TR-94-17, Tucson, AZ: Department of Computer Science, University of Arizona*, 1994
[5] Ricardo A. Baeza-Yates, Gaston H, Gonnet: A New Approach to Text Searching, *Commun ACM*, 35(10), 1992,pp. 74-82
[6] Gonzalo Navarro, Mathieu Raffinot, Fast and Flexible String Matching by Combining Bit-Parallelism and Suffix Automata, *ACM Journal of Experimental Algorithmics*, 2000, 5(4)
[7] Zhou Z, Song T, Jia Y, A high-performance url lookup engine for url filtering systems, *IEEE International Conference on communications*, 2010, pp.1-5
[8] Lin M, Chiu C, Lee Y, Pao H, Malicious URL filtering-A big data application, *IEEE international conference on Big Data*, 2013, pp.589-596
[9] Yuan Z, Yang B, Ren X, Xue Y, TFD: A multi-pattern matching algorithm for large-scale URL filtering, *International Conference on Computing, Networking and Communications*, 2013, pp.359-363
[10] Liu Y, Shao Y, Wang Y, Liu Q, Guo L, A multiple string matching algorithms for large scale URL filtering, *Chinese Journal of Computer*, 2014, pp.1159-1169
[11] Rivest R, The MD5 Message-Digest Algorithm, *RFC Editor*, 1992, 473 (10), pp.492-492
[12] Peterson W W, Brown D T, Cyclic Codes for Error Detection, *Proceedings of the Ire*, 49(1), 1961, pp.228-235]
[13] Liu Y, Ping L, Tan J, Guo L, A Multiple String Matching Algorithm Based on Memory Optimization, *Journal of Computer Research & Development*, 46(10) , 2009, pp.1768-1776.
[14] https://en.wikipedia.org/wiki/Linear_congruential_generator
[15] http://mmix.cs.hm.edu/
[16] https://nuclear.llnl.gov/CNP/rng/rngman/node4.html
[17] http://www.partow.net/programming/hashfunctions/index.html#AvailableHashFunctions
[18] https://en.wikipedia.org/wiki/Birthday_attack