TFD: A Multi-pattern Matching Algorithm for Large-scale URL Filtering

Zhenlong Yuan[*†], Baohua Yang[*†], Xiaoqi Ren[*] and Yibo Xue[†‡]
[*]Department of Automation, Tsinghua University, Beijing, China
[†]Research Institute of Information Technology, Tsinghua University, Beijing, China
[‡] Tsinghua National Lab for Information Science and Technology, Beijing, China
{yuanzl11, ybh07, renxq08}@mails.tsinghua.edu.cn, yiboxue@tsinghua.edu.cn

***Abstract*—During the past decade, URL filtering systems have been widely applied to prevent people from browsing undesirable or malicious websites. However, the key method of URL filtering, such as URL blacklist filter, is more challenging due to the limited performance of existing multi-pattern matching algorithms. In this paper, we propose a multi-pattern matching algorithm named TFD for large-scale and high-speed URL filtering. TFD employs *Two-phase hash*, *Finite state machine* and *Double-array storage* to eliminate the performance bottleneck of blacklist filter. Experimental results show that TFD achieves better performance than existing work in terms of matching speed, preprocessing time and memory usage. Specially, on large-scale URL pattern sets (over 10 million URLs), with single thread, TFD's matching speed reaches over 100Mbps on a general x86 platform.**

***Index Terms*—URL filtering; Blacklist; Multi-pattern matching; Large-scale; Matching speed**

I. Introduction

URL filtering has been widely deployed in several network security devices, such as firewall and IDS/IPS, to protect people from suffering various attacks. Generally, the most direct and effective way of URL filtering is to use multi-pattern matching algorithms in blacklist filter, i.e., take each URL in the blacklist as one pattern, and use the multi-pattern matching algorithm to filter malicious URLs. However, with the rapid growth of malicious websites, URL blacklist filter is more challenging than before due to the following critical requirements:

- *Large scale*: URL blacklist filter should support filtering all the suspicious URLs at one time. But the total number of malicious URLs is humongous. For example, based on the published data [1], the number of them has reached over 3.50 million.
- *High speed*: Considering the large-scale URL pattern set, high speed should also be guaranteed. Slow or unstable speed of URL filtering will affect the overall performance of network system.
- *Low memory consumption*: Due to the memory capacity limitations of real devices, low memory consumption is necessary for URL filtering in practical application.

Looking back upon existing work, we found that although a great number of multi-pattern matching algorithms have been proposed in recent decades, most of them are designed for pattern sets size typically less than 1 million. According to our experiments on real URL pattern sets, the existing best known algorithms cannot meet the critical requirements of large-scale URL filtering.

In this paper, we propose a novel algorithm named TFD for URL filtering. TFD can support dealing with pattern sets of 10 million URLs, and the matching speed is hundreds of times faster than that of the existing algorithms. Main contributions include:

- *An efficient algorithm*: The TFD algorithm uses *Two-phase hash*, *Finite state machine* and *Double-array storage* techniques that achieves both fast matching speed and acceptable memory usage for large-scale URL filtering. Besides, TFD is superior in supporting fast pattern update and thus more suitable for real-time updating of URL blacklist.
- *Performance evaluation*: Experimental results on real-life URL pattern sets show that, even for 10 million URLs, TFD can still achieve more than 100Mbps matching speed with single thread.

The rest of the paper is organized as follows. Section II gives the related work of our research. Section III introduces the key ideas of TFD algorithm. The details and workflow of TFD algorithm are presented in Section IV. Section V evaluates the performance of TFD compared with several existing algorithms. Section VI concludes the paper.

II. Related Work

Along with the wide application of URL filtering, lots of related techniques [2]–[5] are proposed. Moreover, URL blacklist filter accomplished by multi-pattern matching has been proved to be fundamentally efficient for URL filtering. Theoretically, the multi-pattern matching problem can be solved by using strategies based on either finite state machine (FSM) [6]–[10] or hash of character blocks [11]–[14].

Aho et al. showed that FSM can be efficiently used in multi-pattern matching [10]. They proposed the popular AC algorithm which has constant matching speed, high robustness and good expansibility. Aoe came up with the Double-Array algorithm [9], it employed a two-dimensional array (*base* and *check*) to store FSM and achieved good performance in terms of matching speed. However, as the memory usage of FSM can be prohibitively large, the requirement of low memory consumption is difficult to meet, especially for large-scale URL pattern sets.

Hash-based algorithms often use character blocks to achieve low memory usage and satisfied speed. Based on the idea of skip, Wu et al. used hash tables in their WM algorithm [13]. However, when the pattern set becomes larger, it will cause hash collision which greatly reduces the matching performance. One recent study by Zhou proposed MDH algorithm [14], which optimized WM algorithm with multi-phase hash and dynamic-cut heuristics strategies. According to Zhou's experiments, the performance of MDH is superior to WM and some other algorithms. But in our experiments, we observed that it still could not fundamentally solve the problem of hash collision occurred in large-scale URL pattern sets.

Therefore, existing best known multi-pattern algorithms cannot meet all the three requirements for large-scale URL filtering. The FSM-based solutions suffer from memory explosion while the hash-based algorithms cannot efficiently solve the problem of hash collision. In our research, we focus on improving the hash-based solution.

III. Key Ideas

A. Motivation

For the hash-based algorithms, the problem of hash collision is inevitable. On one hand, the size of hash table is limited but the number of URL patterns is huge. On the other hand, the character blocks are non-uniformly distributed, which results in unbalanced sizes of entries in hash table and significantly degrades the matching speed. More importantly, the collision cannot be solved by adjusting hash functions. Based on the above, we employ *Four-byte Block*, *Two-phase hash*, *Finite state machine* and *Skip after Exact Matching* to optimize the WM algorithm. In addition, due to the memory explosion brought by FSM, we absorb the idea of Double-Array algorithm by using *Double-Array Storage*.

B. Key Ideas

We describe all the key ideas in detail as below:

1) Four-byte Block: First, we use B to represent the length of the character block. WM avoids unnecessary matching by skip. Specifically, it uses a two-byte (B=2) character block to distinguish every pattern, which is extremely effective for small pattern sets. But when the pattern set reaches million level, there will be many zero value entries in hash table. Then the matching speed is badly reduced. To solve this problem, we use four-byte character block in TFD instead of two-byte in WM. Experiments proved that the skip possibility is enhanced significantly. Wu and Manber also proved that it was best when $B = log_{|\sum|}(2 * l_{min} * r)$ [13], in which $\sum$ represents the number of possible characters, l_{min} represents the minimum pattern length and r represents the size of the pattern set.

2) Two-phase Hash: Although four-byte character block can improve matching performance, larger B will result in bigger hash tables. Four-byte full address will need 2^{4*8}, i.e., about 4GB space, which will greatly increase the memory requirement.

In order to solve this issue, we use two-phase hash to increase the matching speed and maintain moderate memory consumption. TFD builds two compressed hash tables–SHIFT and MAP table by hash function h_1 and h_2. Assume that h_1 produces a N_1–bit hash value and h_2 produces a N_2–bit hash value. Then SHIFT table has 2^{N_1} entries while MAP table has 2^{N_2} ones. That is, TFD chooses a compressed hash function h_1 that converts the B-length character block (8B bits) to N_1 bits to build the SHIFT table, while uses another compressed hash function h$_2$ to converts the B-length character block (8B bits) to N_2 bits to build the MAP table.

In the matching process, TFD uses SHIFT table to get the skip value and avoid unnecessary operations. Besides, the MAP table can deal with patterns which have zero value in SHIFT table. Therefore, two-phase hash reduces the ratio of entries with zero shift value. Note that "skip value" and "shift value" represent for the same value in the SHIFT table.

3) Finite State Machine: To solve the problem of hash collision in hash entries, we build FSM in every entry of MAP table. By this way, we combine *two-phase hash* with FSM, thus TFD can finish the matching for patterns linked in a certain MAP entry by one-time searching while WM only uses Brute-Force algorithm.

4) Skip after Exact Matching: In WM algorithm, the slide window of input text only slides one byte after finishing exact matching for an entry of the MAP table. As a matter of fact, the window can slide more than that. For example, if m represents the size of slide window, suppose m is 8 and the hashed character block is "*abcd*". Then if this block "*abcd*" never appears in the other patterns, we can directly slide the window 5 bytes (which comes from $m - B + 1$) from left to right; or if this block "*abcd*" only appears in the leftmost four characters, then we can slide the window 4 bytes from left to right, and so on. In this way, it avoids unnecessary matching. Therefore, we add a skip value for every entry of MAP table to skip after exact matching, and further improve the matching performance.

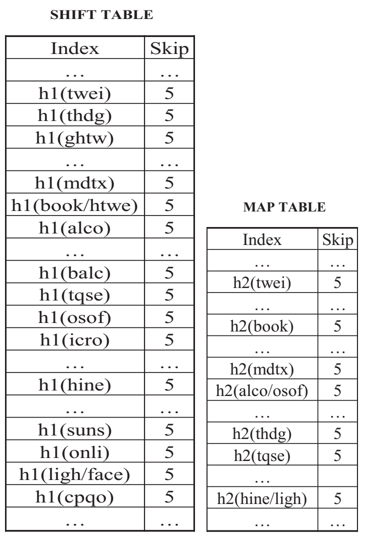
5) Double-array Storage: To solve the problem of memory explosion caused by FSM, we use two-dimensional arrays to store FSMs. As a result, the space complexity reduces from $O(|P|*|\Sigma|)$ to $O(|P|)$ (P represents for the sum of the length of all the patterns). As Double-Array algorithm only uses additive operations to transfer from one state to another while FSM has to handle others like string comparisons and copying, the matching speed is faster than that of FSM in practical use.

The reason why TFD is more suitable for the real-time URL pattern updating can also be explained by this. We mainly need to update the double-array structure which stores the corresponding patterns linked in the MAP entry instead of performing the whole preprocessing again.

The matching performance can be greatly improved by the key ideas mentioned above.

IV. TFD Algorithm

Based on the key ideas described in section III, we propose the novel TFD algorithm. It can deal with pattern sets at ten million's level, and the matching speed is hundreds of times faster compared to the existing algorithms while the memory



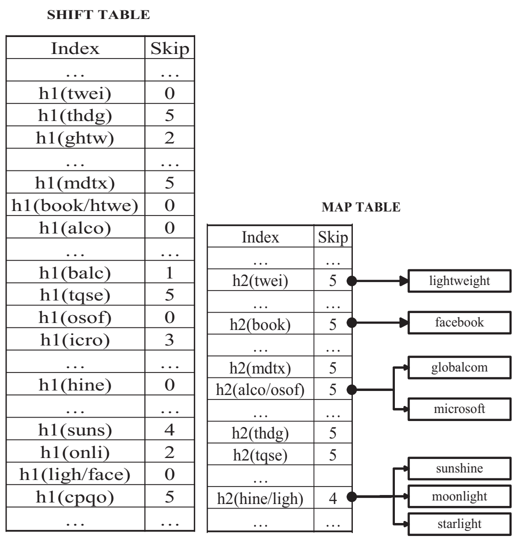
SHIFT TABLE

Index	Skip
…	…
h1(twei)	5
h1(thdg)	5
h1(ghtw)	5
…	…
h1(mdtx)	5
h1(book/htwe)	5
h1(alco)	5
…	…
h1(balc)	5
h1(tqse)	5
h1(osof)	5
h1(icro)	5
…	…
h1(hine)	5
…	…
h1(suns)	5
h1(onli)	5
h1(ligh/face)	5
h1(cpqo)	5
…	…

MAP TABLE

Index	Skip
…	…
h2(twei)	5
…	…
h2(book)	5
…	…
h2(mdtx)	5
h2(alco/osof)	5
…	…
h2(thdg)	5
h2(tqse)	5
…	
h2(hine/ligh)	5
…	…

Fig. 1. Initialization of SHIFT table and MAP table



SHIFT TABLE

Index	Skip
…	…
h1(twei)	0
h1(thdg)	5
h1(ghtw)	2
…	…
h1(mdtx)	5
h1(book/htwe)	0
h1(alco)	0
…	…
h1(balc)	1
h1(tqse)	5
h1(osof)	0
h1(icro)	3
…	…
h1(hine)	0
…	…
h1(suns)	4
h1(onli)	2
h1(ligh/face)	0
h1(cpqo)	5
…	…

MAP TABLE

Index	Skip
…	…
h2(twei)	5
…	…
h2(book)	5
…	…
h2(mdtx)	5
h2(alco/osof)	5
…	…
h2(thdg)	5
h2(tqse)	5
…	
h2(hine/ligh)	4
…	…

lightweight
facebook
globalcom
microsoft
sunshine
moonlight
starlight

Fig. 2. The results of SHIFT table and MAP table

requirement stays in a comparatively low level. Moreover, TFD is easy to be implemented. In this section, we present TFD in detail based on the procedures of preprocessing and matching.

A. Preprocessing

Compared with other algorithms, TFD finishes more operations in preprocessing. We take the pattern set {*lightweight, facebook, globalcom, microsoft, sunshine, moonlight, starlight*} for example, the preprocessing is as follows:

1) Read all the patterns in sequence and store them. Initialize the related information.

2) Determine the hash functions–h_1 and h_2 according to the pattern sets scale, platform and cache size. Suppose that $N_1 = 26$, $N_2 = 23$, then the hash functions are:

$h_1(block) = (*(block))\&0x03FFFFFF$

$h_2(block) = ((*(block) \ll 15) + (*(block+1) \ll 10) + (*(block+2) \ll 5) + *(block+3))\&0x007FFFFF$

3) Initialize the SHIFT and MAP table, and assign the value m-B+1 (m is the size of slide window and B is the size of character block) to every entry. The result is shown in Fig.1.

4) Perform two-phase hash for all the character blocks. Store the skip value into the entry of SHIFT table and link the patterns which have the same hash value into the corresponding entry of MAP table. Then compute and store the skip value for every entry of the MAP table. SHIFT table and MAP table for the pattern set mentioned above are shown in Fig.2.

5) Build FSM for the patterns linked in an entry of MAP table, and then all the patterns of an entry make up a trie. For instance, the FSM which is built for h2(hine/ligh) in the MAP table above is shown in Fig.3.

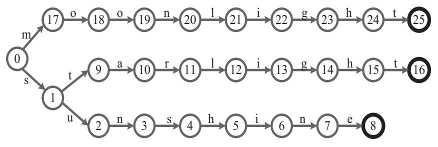


Fig. 3. FSM (Finite State Machine)

6) Store the trie in a two-dimensional array, that is to replace the FSM built in procedure 5 with double-array. After that TFD frees the memory of the FSM. The details of double-array storage are described as below.

The structure of the two-dimensional array includes two values: *base* and *check*. *Base* represents for a trie node and *check* points to the last state. If *base* and *check* are both 0, it means that this state is null. If the *base* is negative, then there is a match happened in this state. We use t to represent the current state, s to represent the state before t and c to represent the input character, then $check[base[s] + c] = s$ and $base[s] + c = t$.

The process includes two main steps: firstly, encode all the characters that may appear, the encoding result for characters is shown in TABLE I; secondly, construct the double-array by recursive algorithm, the result is shown in TABLE II.

TABLE I
Encoding result for characters

character	s	u	n	h	i	e	m	o	l	g	t	a	r
encoding	1	2	3	4	5	6	7	8	9	10	11	12	13

7) Repeat procedure 5 and 6 for the next MAP entry until it finishes the double-array storage for all of the entries.

B. Matching

After finishing the preprocessing, it can be moved to matching stage. The matching process with pseudo code is shown in Algorithm 1.

TABLE II
THE DOUBLE-ARRAY RESULT

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Base	1	1	1	4	3	6	11	8	2	3	3	1	1	2	12	16	10	17	16	19	13	-1	14	-1	-1
Check	0	0	1	3	4	10	5	11	2	9	14	1	12	13	6	11	15	8	16	17	18	19	20	21	23
State	s	m	su	sun	suns	moon	sunsh	starli	mo	moon	starl	st	sta	star	moonl	sunshi	moonli	starlig	sunshin	moonlig	starligh	sunshine	moonligh	starlight	moonlight

Algorithm 1 Matching Process of TFD Algorithm

Require: $offset = text + m - B$, $end = text + textlen - B$. BFCompare represents matching using BF algorithm, DACompare represents matching using double-array.

```
while $offset \le end$ do
    $value \leftarrow$ h1($offset$);
    $skip \leftarrow$ shift[$value$];
    if $skip = 0$ then
        $value \leftarrow$ h2($offset$);
        $skip \leftarrow$ hash[$value$].$skip$;
        $num \leftarrow$ hash[$value$].$num$;
        if $num \ge 0$ then
            if $num = 1$ then
                BFCompare($offset$, hash[$value$].$pat$);
            else
                DACompare($offset$, hash[$value$].$arr$);
            end if
        end if
    end if
    $offset \leftarrow offset + skip$;
end while
```

V. Evaluations and Discussions

In this section, we evaluate and compare performance of TFD with other best known multi-pattern matching algorithms, such as AC, Double-Array, WM and MDH in terms of matching speed, memory usage and preprocessing time. These metrics are measured by pattern sets from small to large scale with the same matching rate. Besides, we also use pattern sets which have different hash collision degrees to compare the matching speed variation of these algorithms.

A. Data-set and Test-bed

We first present how to get the data sets for our experiments. As the amount of published malicious URLs is just 3.50 million, far less than our requirement. Thus the URL data sets used in our experiments are all obtained from network crawler. Specifically, we obtain two large data set: *URL A* (2.08G) and *URL B* (3.4G).

1) ***Input Text Set****:* *URL A* is used as the input text set.

2) ***Pattern Sets I****:* *Pattern sets I* are used in the first three experiments. By integrating *URL A* and *URL B*, we get pattern sets from 10K to 10 million but with the same matching rate of 10%.

3) ***Pattern Sets II****:* *Pattern sets II* are used in the fourth experiment. They are obtained from *pattern sets I*, and have the scale of 50K URLs. In addition, *pattern sets II* have the same matching rate but different degrees of hash collision.

4) ***Test Platform****:* The experiments are running on the platform: the processor is Intel® Xeon® CPU E5504 running at 2GHz; the memory is 8GB (DDR3 800MHz) and the Smart Cache is 4MB; the operating system is CentOS Linux release 6.0 (Final). For the aim of fairness, in performance comparison, all the algorithms are tested on a single thread.

B. Experimental Results

1) ***Matching speed vs. Pattern sets scale****:* Matching speed for different scale pattern sets is illustrated in Fig.4. From this figure, we can see that the matching speed of WM and MDH declines rapidly with the increase of pattern sets scale, whereas TFD's matching speed only decreases linearly. When the pattern sets scale is larger than 50K, AC and Double-Array cannot work because of memory explosion, and WM's matching speed is 60% less than that of TFD. For the scale of ten million, the matching speed of TFD reaches 13.1MB/s, which is 217.5 and 600 times faster than WM and MDH respectively.

2) ***Memory storage vs. Pattern sets scale****:* From Fig.5 we can see the memory storage of AC and Double-Array algorithm is almost identical as they are all based on FSM. For the pattern sets scale between 10K and 50K, TFD consumes 4.5% memory storage compared with that of AC or Double-Array algorithm. However, as the pattern sets become larger, the memory storage they consume increases extremely fast. TFD respectively requires 2GB and 4GB memory for pattern sets with 5 and 10 million signatures. In comparison, MDH and WM consume 1GB memory storage on the 10 million pattern set. Fortunately, as the computer hardware develops, it is cheap enough to buy servers with large memory such as 8GB or 16GB.

3) ***Preprocessing time vs. Pattern sets scale****:* Preprocessing time of AC, Double-Array, WM, MDH and TFD is shown in Fig.6. We can see that all the experimented algorithms consume very little time in preprocessing if the pattern set is less than 2 million. But when the scale is larger than 2 million, the preprocessing time of TFD and MDH tends to rise. In fact, we just preprocess once in practical use, hence it is acceptable to preprocess for dozens of minutes especially for large-scale pattern sets.

4) ***Matching speed vs. Hash collision****:* Fig.7 shows how the matching speed of these algorithms changes when handling pattern sets which have different degrees of hash collision. The hash collision degree increases from *Data 1* to *Data 4*. We notice that TFD has the most efficient performance

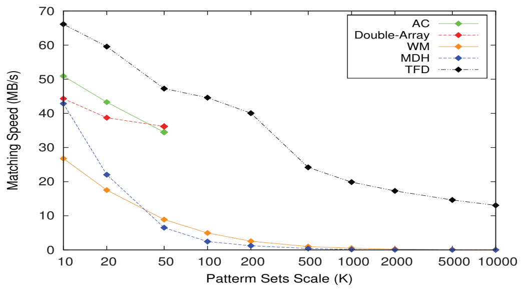


Fig. 4. Matching speed vs. Pattern sets scale

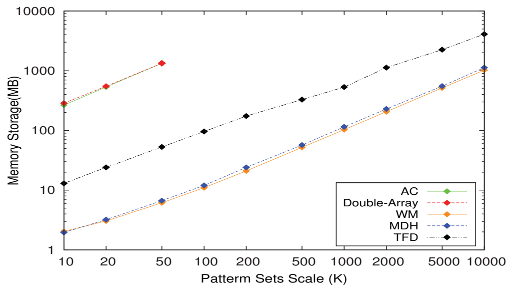


Fig. 5. Memory storage vs. Pattern sets scale

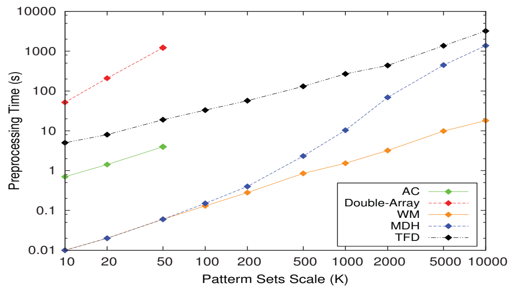


Fig. 6. Preprocessing time vs. Pattern sets scale

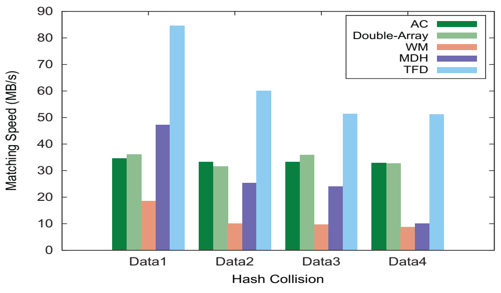


Fig. 7. Matching speed vs. Hash collision

compared with other algorithms under any hash collision degree. However, the speed of WM and MDH algorithm declines rapidly when hash collision becomes more severe. In the worst case, it declines to 20% of the speed of TFD algorithm. In comparison, TFD's speed tends to be stable in matching process as the degree of hash collision changes.

C. Discussion

Theoretically, the advantage of AC algorithm is that it has linear matching speed, while WM algorithm can skip by building the hash table. TFD not only gathers the two superiorities above, but also solves both the problem of memory explosion and hash collision. This explains why it has more superior performance than AC, Double-Array, WM and MDH for large-scale URL filtering.

Crucially, the most significant superiority of TFD algorithm is the high matching speed which is also the most concerned performance metric in practical use. Furthermore, the memory storage problem can be solved by the improvement of computer hardware.

VI. Conclusion

The key method of URL filtering, such as URL blacklist filter, is more challenging due to the limited performance of existing multi-pattern matching algorithms. In this paper, we propose a novel multi-pattern matching algorithm named TFD for large-scale URL filtering. TFD combines the ideas of *Two-phase hash*, *Finite state machine* and *Double-array storage*. Experimental results on real-life URL pattern sets show that, even for 10 million URLs, TFD can still achieve more than 100Mbps matching speed with single thread. This performance can also be improved by parallelism.

Acknowledgment

This work was supported in part by National High-Tech R&D 863 Program of China under grant No. 2012AA010905, and Tsinghua National Laboratory for Information Science and Technology (TNList) Cross-discipline Foundation.

References

[1] *URL Blacklist*. [Online]. Available: http://urlblacklist.com/

[2] Y. Feng, N. Huang, and C. Chen, "An efficient caching mechanism for network-based url filtering by multi-level counting bloom filters," in *Proc. of IEEE ICC*, 2011.

[3] L. Chou, Z. He, D. Li, H. Chen, J. Su, C. Chen, H. Wei, and C. Li, "Design and implementation of content-based filter system on embedded linux home gateway," in *Proc. of IEEE ICACT*, 2012.

[4] Z. Zhou, T. Song, and Y. Jia, "A high-performance url lookup engine for url filtering systems," in *Proc. of IEEE ICC*, 2010.

[5] *ufdbGuard*. [Online]. Available: http://www.urlfilterdb.com/

[6] N. Hua, H. Song, and T. Lakshman, "Variable-stride multi-pattern matching for scalable deep packet inspection," in *Proc. of IEEE INFOCOM*, 2009.

[7] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. of IEEE INFOCOM*, 2004.

[8] A. Bremler-Barr and Y. Koral, "Accelerating multi-patterns matching on compressed http traffic," in *Proc. of IEEE INFOCOM*, 2009.

[9] J. Aoe, "An efficient digital search algorithm by using a double-array structure," *Software Engineering, IEEE Transactions on*, vol. 15, no. 9, pp. 1066–1077, 1989.

[10] A. Aho and M. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[11] B. Xu, X. Zhou, and J. Li, "Recursive shift indexing: a fast multi-pattern string matching algorithm," in *Proc. of ACNS*, 2006.

[12] D. Oh and W. Ro, "Multi-threading and suffix grouping on massive multiple pattern matching algorithm," *The Computer Journal*, 2012.

[13] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Technical Report TR-94-17, University of Arizona, Tech. Rep., 1994.

[14] Z. Zhou, Y. Xue, J. Liu, W. Zhang, and J. Li, "Mdh: A high speed multi-phase dynamic hash string matching algorithm for large-scale pattern set," *Information and Communications Security*, pp. 201–215, 2007.