Towards a Fast Regular Expression Matching Method over Compressed Traffic

Xiuwen Sun, Hao Li, Xingxing Lu, Dan Zhao, Zheng Peng, Chengchen Hu
Department of Computer Science and Technology
Ministry of Education Key Lab for Intelligent Network and Network Security
Xi'an Jiaotong University

***Abstract*—Nowadays, Deep Packet Inspection (DPI) becomes a critical component of the network traffic detection applications. For comprehensive analysis of traffic, regular expression matching as the core technique of DPI is widely used. However, web services tend to compress their traffic for less data transmission, which challenges the regular expression matching to achieve wire-speed processing. In this paper, we propose *Twins*, a fast regular expression matching method over compressed traffic that leverages the returned states encoding in the compression to skip the bytes to be scanned. In our evaluation results, Twins can skip about 90% compression data and can achieve 1.5Gbps throughput, which gains 2.7∼3.4 performance boost to the state-of-the-art work.**

***Index Terms*—Deep Packet Inspection, Regular Expression Matching, Multi-Pattern Matching, Compressed Traffic.**

I. Introduction

Deep Packet Inspection (DPI) technique has been promoted from the simple string matching to semantics-based analysis to better serve the emerging scenarios including network optimization, security, big data analysis [1], [2], *etc*. In particular, the regular expression (RegEx) can describe higher-level semantics than plain string, and therefore its matching methods become vital for realizing a DPI system.

However, today's web server tends to compress their contents beforehand, so as to reduce the transmission overhead for improving the user experience. It heavily impacts traditional RegEx matching, *aka*, *Naive* method, that only works for the uncompressed content. To be specific, due to the 20% compression ratio of the compressed traffic [3], [4], it requires a 5× speed-up for Naive method to maintain its original performance.

In general, compressed traffic matching (CTM) consists of two independent stages: decompression and matching. The first stage is fast enough and not critical [5], so the second stage determines the performance of the whole process. The principle of accelerating the matching stage is to leverage the hidden information of the compressed traffic, *e.g.*, the flags added by the compression algorithm, according to which many of the traffic can be skipped with a quick glance of the pre-stored information. Note that the processing on the hidden information will also bring extra cost. As a result, the performance of CTM depends on two factors: (1) the number of bytes that can be skipped, and (2) the extra cost for identifying such bytes, both of which can be largely improved for previous works. For example, the state-of-the-art work ARCH [6] skips 79% of the traffic that Naive scanned, while theoretically more than 90% bytes encoded in compression segments can be skipped according to the analysis in Section IV. Besides, the previous works incur large extra cost, so they can barely achieve the wire-speed processing. For example, ARCH only achieves 450Mbps throughput according to our evaluation.

Therefore, there is high potential to increase the performance of CTM by skipping more bytes with less extra cost. In this paper, we propose *Twins* method to achieve the above goals. The basic idea is to leverage the *locality principle* lying in compression format, *i.e.*, the hidden information of scanning the identical strings are mostly the same, so that Twins can skip more bytes by just checking the previous scan results.

To be specific, the contributions of this paper are:

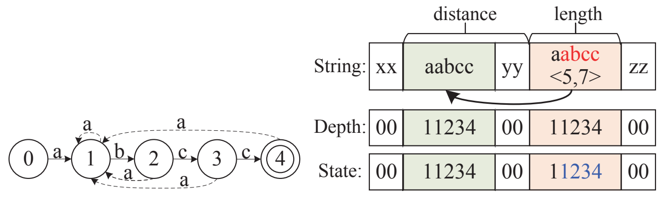
1) We propose a model to evaluate the performance of CTM by considering the aforementioned two factors to reveal the design space of CTM.
2) We present a novel method Twins to achieve the wire-speed RegEx matching for CTM.
3) We prototype Twins, which can achieve more than 1.5Gbps throughput, bringing a 2.7∼3.4× performance boost to ARCH.

II. Design Space of RegEx Matching over Compressed Traffic

A. Design Principle

As mentioned, the basic principle of accelerating CTM is to leverage the hidden information of compressed data and skips the bytes of pointer when matching the pattern(s). It is achievable because the traffic is compressed by minimizing the redundant contents, and therefore such information of redundancy can be utilized for fast skipping.

Specifically, more than 90% of the web sites use gzip [7] as their default compression encoding format [4]. gzip uses DEFLATE [8] as its compression method, which is a combination of the LZ77 [9] algorithm and Huffman coding. During compression, LZ77 tries to find repeated maximum occurrences substring with references to the earlier data by employing a sliding window and replace them by a pair of $< length, distance >$, where *length* is the length of the substring and the *distance* is the distance between the substring and the corresponding reference. To unify the terms, the pair



(a) The DFA constructed by "abcc" (b) Inspected string and hidden info.

Fig. 1. The example of DFA and inspected string with corresponding stored hidden information. The substring "aabcc" between 'y' and 'z' is coded as <5,7>, which represents a pointer, and the bytes not in pointer are literals. State line stores the returned states by scanning the bytes above them. Depth line represents the parameter depth.

is called *pointer* and the corresponding substring is called *referred string*. The positional relation of the terms is shown in Fig. 1(b). After that, the compressed data, which contains literals and pointers, are encoded by Huffman coding.

In our survey, we find the pointer ratio (the ratio of bytes represented by pointers) is more than 91%. For example, in Fig. 1(b), the uncompressed traffic size is 16B and the compressed traffic size is 13B by encoding the second substring "aabcc" to a pair of "<5,7>". Only 5 bytes represented by pointer, thus, the pointer ratio is $5/16$ and the compression ratio is $13/16$.

B. Performance Evaluation Model

For RegEx matching based on finite state automaton (FSA), it usually compiles RegExs to a non-deterministic finite automata (NFA) first. Then, the NFA is converted to a corresponding deterministic finite automata (DFA) and the DFA is minimized. At last, it finds patterns accepted by this DFA.

Consider a simple example shown in Fig. 1. The DFA is constructed by string pattern "abcc". The parameter Depth is the shortest length from the current active state to the root state of DFA and could represent the length of pattern's prefix. We keep it as the hidden information during the process of scanning. While scanning the string "aabcc" in pointer, we get the depth of the last pointer byte first, *i.e.*, 4. Then, the matching goes backwards 4 bytes and starts from the second 'a'. It would find the pattern "abcc" and skip the scanning of the first 'a'. So, the time of matching one byte can be saved and the time of processing and indexing the stored parameter depth becomes the extra cost. If the extra cost is low enough, the throughput of CTM can be improved.

Formally, we use t_s to denote the time consumption of the process that one byte is matched by FSA. For a certain amount of transmitted traffic bytes D (compressed or uncompressed), we can get the throughput of uncompressed matching as $T_u = D/Dt_s = 1/t_s$.

We use t_c to denote the extra cost time for skipping one byte, R_c to represent the compression ratio and R_s to denote the skipped ratio, which is the ratio of skipped bytes using by accelerated algorithm out of the decompressed traffic. Obviously, the maximum of skipped ratio equals to pointer ratio for CTM methods, but $R_s = 0$ for naive method. As a result, the throughput of CTM can be calculated as follow.

$$\begin{aligned} T_c &= \frac{D}{t_s(1-R_s)D/R_c + t_c R_s D/R_c} \\ &= \frac{R_c}{(1-R_s)t_s + R_s t_c} \end{aligned} \tag{1}$$

The parameter R_c relies on the characteristics of traffic and R_s is determined by the CTM methods, both of which can be regarded as the fixed values. So, to improve the processing throughput, the method is expected to minimized t_s and t_c.

C. Limitations of Prior Works

Now we could evaluate the existing CTM methods by using the above equation and obtain an ideal throughput improvement compared to Naive method.

Naive method skips nothing and brings no extra cost, so $R_s = 0, t_c = 0$ and its throughput can be calculated as $T_{naive} = R_c/t_s$ according to Eq. (1). Compared with the uncompressed matching, the throughput of Naive method is only determined by the compression ratio of the processed traffic. With the 20% compression ratio as we mentioned before, the throughput of Naive method is only 20% of the uncompressed matching's.

For any other methods, the throughput promotion than Naive method can be described as Eq. (2). So, the promotion is determined by R_s and t_c/t_s, namely the ratio of skipped bytes and the extra cost of skipping one byte to the time consumption of matching this byte. While $t_c = 0$, we can get an ideal promotion that $P_{ideal} = 1/(1-R_s)$. Thus, $P_{ideal} = 10$ with 90% of the pointer ratio and $P_{ideal} = 5$ with 80% of the pointer ratio.

$$\begin{aligned} P &= \frac{T_c}{T_{naive}} = \frac{t_s}{(1-R_s)t_s + R_s t_c)} \\ &= \frac{1}{1-(1-t_c/t_s)R_s} \end{aligned} \tag{2}$$

A method would get a significant performance boost with a larger R_s and a smaller t_c/t_s. It may approach wire-speed processing rate only when t_s is small enough either. ARCH has skipped 79% bytes of the decompressed traffic, which is not enough for 91% pointer ratio. Furthermore, its overhead is so high for calculating Input-Depth and incurring a redundant process in its algorithm that the throughput of matching is not enough to meet the requirement of the wire-speed. Therefore, we propose a method Twins to pursue the two goals and elaborate the detail in the next section.

III. Design of Twins

A. Basic Idea

Following the above analysis, one key point of further improving the performance of CTM is to minimize the extra cost when skipping the input. In other words, skipping a byte must be faster than scanning it. However, the modern DFA implemented by the 2-dimensional array transition table can scan a byte with only a few memory accesses, and it is extremely difficult, if not impossible, to take even fewer memory accesses for skipping a input byte.

In contrast, Twins attempts to improve the efficiency of the skipping process, *i.e.*, skipping more bytes by only checking one, so that the total overhead can be reduced. This is possible due to the following two observations: (1) the content of a pointer and its referred string are identical, and (2) given an input byte, it is very possible that the next state is the same state, regardless of the current state.

Based on the two observations, we can make a hypothesis of locality principle for compressed data that *in most cases, the returned states are the same as the referred strings and their pointers*. Therefore, there lies a chance that using returned states as the hidden information. The bytes in pointer can be skipped, while finding the current returned state in pointer is equal to the previous stored one.

For example, in Fig. 1(b), the returned states are stored as the hidden information. When DFA begins to scan the bytes in the pointer, the current active state is “0” and equals to the stored state before its referred string. It will return the same states (“11234”) of scanning the bytes (“aabcc”) in the referred string and pointer. Because DFA returns the same state when it scans the same string with a same active state. Thus, the scanning of bytes in this pointer can be skipped and the scanning of bytes followed the pointer can be continued with the stored state (‘4’) in referred string.

B. Algorithm

Twins inspects the uncompressed data after decompression and stores the returned states as hidden information. After that, it accelerates inspection of pointer bytes with the help of the stored states. The function *CompressedMatching* in Algorithm 1 shows the CTM routine on inspecting literals and pointer bytes in traffic. It invokes FSA procedure to check each literal and processes pointer bytes by Twins.

The locality principle is just like a pair of twin babies, most of their characteristics are same, thus we call the method Twins. The correctness of Twins can be supported by Theorem 1 in textbook [10], which is described as follow:

Theorem 1. *Given a FSA $A = (Q, \Sigma, \delta, q_0, F)$, $\hat{\delta}$ is the extended transition function. For any symbol $a \in \Sigma$ and string $w \in \Sigma^*$, if $p = \hat{\delta}(q, w)$, then $\hat{\delta}(q, wa) = \delta(p, a)$.*

Informal description is that the next state of FSA is determined by current state and the next input symbol, without the influence of previous scanned string.

According to Theorem 1, if the returned states of scanning the bytes before pointer and referred string are equal. All the returned states of scanning pointer bytes must be same to the states of scanning referred string. For example, if $q_k = q_m$ in Fig. 2, then $q_x = \hat{\sigma}(q_k, w_0, \cdots, w_n a_x) = \hat{\sigma}(q_m, w_0, \cdots, w_n a_x) = \sigma(q_s, a_x)$. It could continue the scanning with q_j and all of the pointer bytes would be skipped without re-scanning.

When scanning a byte in pointer, if the returned state equals to the state with the same offset in its referred string, it can skip pointer bytes behind this position. As in Fig. 2, we assume $q_k \neq q_m$. The method continues to scan the pointer bytes and

String:	a_m	$w_0 \cdots w_n$	$\cdots a_k$	$w_0 \cdots w_n$	a_x
State:	q_m	$q_r \cdots q_s$	$\cdots q_k$	$q_i \cdots q_j$	q_x

Fig. 2. Inspected data and stored states for Twins.

Plaintext: xxabbabcdyyabbzzaabccd
Compression: xxabbabcdyy<3,9>zza<3,12>

Fig. 3. Input data of example. There are two colored <length, distance> pairs in the compressed data to represent pointer.

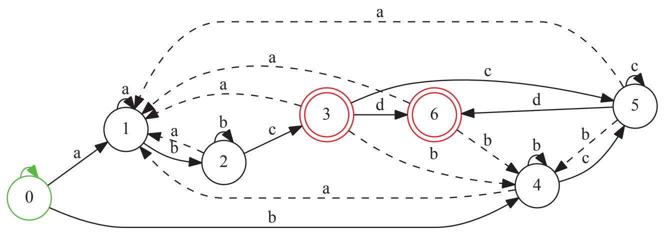


Fig. 4. DFA for ‘$(ab + c) \mid (bc + d)$’. The green circle state is start state and double-circle states are the accepting state. We omit transitions leading to state 0 for convenience.

Ex.1	xx	abb	ab···yy	abb	zz
State:	00	122	12···00	122	00

(a)

Ex.2	bb	abc	dy···za	abc	cd
State:	22	123	60···01	123	56

(b)

Fig. 5. Examples of matching process of Twins. State line represents returned states of scanning the bytes above them. Red strings are matched patterns and blue states are copied from the referred string to the pointer.

get $q_i = q_r$ after scanning the first byte w_0 in pointer. So, the scanning of the following bytes behind w_0 can be skipped.

Therefore, Twins stores the states returned by FSA while checking each byte. When processing the pointer bytes, Twins continues to scan them until finding the returned state equals to the stored one at the same offset in its referred string. At this time, Twins stops scanning and copies the states from the referred string to the pointer behind this position. If there are any accepting states, Twins records the state and position as matching information. At last, it continues to scan the byte following pointer with q_j (in Fig. 2) as the current state.

The detail of Twins is shown as Algorithm 1. We will elaborate the algorithm with an example in the next.

C. Examples

We use the compressed data in Fig. 3 as input traffic and the DFA in Fig. 4 to present the details of Twins on matching pointer bytes. There are two pointers in this example and we denote them as $P1$ and $P2$.

Each sub-figure in Fig. 5 corresponds to the matching process of the two pointers. The first line is part of decompressed traffic and the second line represents states of checking the bytes above them.

Ex.1: The state (‘0’) at the position in front of $P1$ is equal to the one (‘0’) in front of its corresponding referred string. So, Twins only needs to copy states (‘122’) of the referred string to $P1$, and checks the accepting state during the process of copying. There is no pattern in $P1$, it returns the last state (‘2’) and skips scanning of 3 bytes.

Algorithm 1: Twins Method

```
definition : $byteInfo$ - {symbol, state}
             $byteList$ - array of $byteInfo$
             $len$ - length of pointer
             $dist$ - distance between referred to pointer
             $state$ - state of checking prior pointer byte
input      : $Trf_1...Trf_n$ - compressed traffic

function CompressedMatching($Trf_1...Trf_n$)
    $curState \leftarrow q_0$;
    for $i \leftarrow 1$ to $n$ do
        if $Trf_i$ is not $pointer(len, dist)$ then
            //scan literals in traffic
            $curState$=FSAScanByte($Trf_i, i, curState$);
            $byteInfo.state = curState$;
            $byteList.Add(byteInfo)$;
        else
            //scan pointer by Twins algorithm
            $byteList.Add(byteList[i - dist : i - dist + len])$;
            $curState$=TwinsScan($i, len, dist, curState$);

function TwinsScan($i, len, dist, state$)
    $offset \leftarrow (i - dist)$;
    for $curPos \leftarrow 0$ to $len$ do
        //scan bytes in pointer
        $pos \leftarrow i + curPos$;
        if $state == byteList[offset+curPos-1].state$ then
            //same position has same state in
              pointer and referred string
            for $k \leftarrow curPos$ to $len$ do
                $byteList[i + k].state =$
                  $byteList[offset + k].state$;
                if $FSA.accept(byteList[i + k].state)$ then
                    Record $byteInfo.state$ and $i + k$;
            return $byteList[i + len - 1].state$;
        else
            $state$=FSAScanByte(
              $byteList[pos].symbol, i, state$);
            $byteList[pos].state = state$;

    //have scaned the whole bytes of pointer
    return $state$;

function FSAScanByte($symbol, i, state$)
    $state = FSA.lookup(state, symbol)$;
    //store a matching record
    if $FSA.accept(state)$ then
        Record $state$ and $i$;
    return $state$;
```

Ex.2: The state ('1') at the position in front of $P2$ is not equal to the one ('2') before its referred string. Twins would have to scan the bytes in $P2$ and return a state ('1') of scanning the first byte, which is same to the stored state on the same offset in referred string. So, it copies the following states ('23') of the referred string to $P2$ and finds a matched pattern "abc" by checking the copied states. Then, Twins returns state "3" and continues to scan the bytes behind $P2$. After that, pattern "abccd" would be found while scanning the following bytes. Twins skips scanning of 2 bytes on processing $P2$.

TABLE I
CHARACTERISTICS OF EXPERIMENTAL DATA SETS

	Alexa.com	**Alexa.cn**
Count of Pages	434	13747
Compressed Size (MB)	15.54	226.95
Decompressed Size (MB)	70.24	1190.99
Pointer ratio	91.21%	91.92%
Average pointer length (B)	14.89	19.84

D. Discussion

- Estimation of extra memory consumption

Twins accelerates the speed of matching by keeping returned states as the hidden information. It only need 32K-entries for storing them, because the maximum distance between the pointer and its referred string is 32768 B, which is specified by DEFLATE. It is enough to store states by 32K×4B memory, which would present more than 4 billion states for DFA. So, the requirement of memory space is scale invariant for any DFA scanning engine.

Compared with hundreds or thousands million bytes memory usage of DFA, thousands-bytes extra cost for Twins is more insignificant. Moreover, it also needs to keep some parameters in ARCH method and its DFA states, such as *status, Input-Depth, Simple/Complex state*. Twins takes less overhead than ARCH on the extra memory consumption and the comparison of them are shown in Section IV.

- Estimation of extra time consumption

From Algorithm 1 and the example above, we can learn that Twins utilize copying states instead of scanning these bytes again. To be simplified, it only needs twice memory accesses which is smaller than the consumption of FSA scanning one input symbol. This is the reason why Twins could obtain better performance than the previous works.

- Processing on encrypted traffic

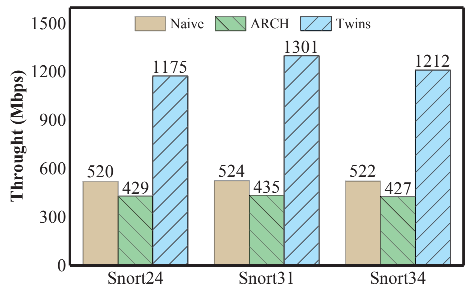
Today's web traffic are often encrypted to protect the data confidentiality and integrity and there are some works [11], [12] that focus on DPI over encrypted traffic. Twins does not concern the matching of encrypted traffic. But, it can be embedded in the systems on processing decryption traffic.

IV. Evaluation

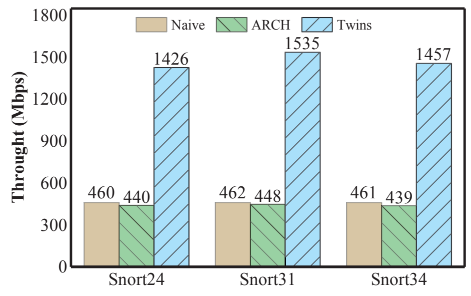
A. Settings

Before evaluation, we have collected traffic by accessing the Alexa top sites shown in Table I and can be access in [13]. Especially, all the raw traffic data are compressed which are collected through browsing home pages of Alexa.com [14] TOP 500 sites and Alexa.cn [15] top 20000 sites, used as the input in our experiments.

To have an intuitionistic comparison with ARCH, we implement Twins over RegEx processor at [16]. The processor provides a DFA-based implementation to perform RegEx matching with an organization of 2-dimensional matrix transition table. It is also used as a base-line algorithm to exhibit the Naive method. We also implement the *Simple/Complex*



(a) Alexa.com data set



(b) Alexa.cn data set

Fig. 6. Evaluation results over two data sets and three RegEx sets.

method in ARCH and that is the only algorithm which have been evaluated by ARCH.

With the same reason for intuitionistic comparison, we also take three RegEx sets, *i.e.*, the Snort24, Snort31, Snort34, which were taken from Snort and published at [16]. Then, a same server (Xeon E5-2630 2.2GHz and 64G RAM) is used for evaluation. All the implementations are single-threaded programming and running over a single core.

B. Performance

Twins and ARCH have matched the same number of patterns as Naive method does which is shown in Table III. Fig. 6 shows the throughput of them over the two sets of compressed traffic and three RegEx sets. Obviously, Twins is more efficient than ARCH and Naive in all the data and RegEx sets. Compared to Naive and ARCH, it achieves 2.3$\sim$3.3 times and 2.7$\sim$3.4 times performance improvement respectively. Moreover, the throughput of Twins has achieved more than 1.5Gbps, that means there is potential to perform wire-speed RegEx matching over compressed traffic with a parallel implementation.

Considering the evaluation model in Eq 2, t_c of Twins is smaller than t_s, which leads to a smaller value of t_c/t_s. It gives Twins a good performance improvement compared to Naive. But, it is not possible for ARCH because the extra cost of skipping pointer bytes is larger than scanning them. ARCH would perform better than Naive in some scenarios that the scan engine is implemented by compression DFA, such as D^{2}FA [17], A-DFA [18]. In these cases, the t_s may be larger than the t_c of ARCH, while the throughput of matching would be reduced.

At last, we evaluate the memory overhead of the matching engines that are implemented by the three RegEx sets. The results show in Table II. From the table, we can find that Twins

TABLE II
MEMORY SIZE OF THE SCANNING ENGINES BY THREE REGEX SETS

RegEx Set	Naive (KB)	ARCH (KB)	Twins (KB)
Snort24	11360	11380	11360
Snort31	6872	6892	6872
Snort34	12676	12688	12676

TABLE III
SKIPPED RATIO OVER THE TWO DATA SET

	RegEx Set	Patterns	ARCH (%)	Twins (%)
Alexa.com	Snort24	32886	78.29	89.69
	Snort31	0	78.10	90.82
	Snort34	259	78.15	89.70
Alexa.cn	Snort24	558514	81.76	91.21
	Snort31	0	82.01	91.81
	Snort34	18059	81.60	91.22

incurs little extra memory than Naive and the size of the extra memory is smaller than ARCH's. Compared to the memory size of Naive, the extra memory used by Twins or ARCH is negligible. Moreover, it is a fixed value for a specific matching engine and distinguishes from the traffic size.

C. Analysis

For quantitative analyzing the evaluation results, we compare Twins and ARCH with another indicator, *skipped ratio* (R_s). Table III shows the skipped ratio of them on processing the traffic with various RegEx sets. It is clear that Twins skips more bytes than ARCH in all the sets. Particularly, Twins skips about 90% bytes on both data sets and almost approaches the upper limits which can be calculated by the pointer ratio of the two data sets (91.35% and 91.92%, shown in Table I).

Moreover, compared the skip ratio and throughput of ARCH over Alexa.cn data set to Twins over Alexa.com data set, there is small difference on the skip ratio, but large difference on their throughput. So, we can find that ARCH gains more extra cost for skipping pointer bytes than Twins with approximately the same skip ratio. Actually, ARCH has to calculate *Input-Depth* parameter and mark *Check, Uncheck* and *Match* flag as the status for each scanned byte, which spends more time than Twins. Therefore, the more bytes are skipped and the lower extra cost is gained, the higher performance will be achieved.

V. RELATED WORK

A. With gzip/DEFLATE

ACCH [3] is based on the Aho-Corasick algorithm [19] for compressed traffic matching and skips matching partial bytes when the pointer does not contain complete pattern. However, if not, it has to scan these bytes again. SPC [20] employs the same basic idea as ACCH to accelerate multi-string matching over compressed traffic for Wu-Manber algorithm [21]. COIN [4] eliminates the redundant process of ACCH and gets better performance than ACCH. SOP [22] was proposed to reduce the memory usage on pattern matching after decompressing traffic, however the speed is relatively lower than even ACCH without any optimization on cutting the matching redundancy. The two papers studied in [23] and [24] aims to match Huffman-encoded data, but they only applied to single-pattern matching rather than multi-pattern matching.

All the methods above are concerned to accelerate string matching over compressed traffic. Except for them, ARCH [6] provides RegEx matching and are described before and thus is omitted here. Sun [25] presents a method which is the first study to perform RegEx matching over compressed traffic. However, it relies on the compression DFA which have reduced its number of path pairs significantly. That limits its using scenarios.

B. With other compression methods

The paper [26] achieved multi-pattern matching over compressed data with LZW, but it cannot work on LZ77 compression algorithm and thus cannot support inspections over HTTP traffic. In [27], the authors applied the Boyer-Moore algorithm [28] to compressed traffic as well as the pattern for fast matching. However, the compressing method is also a single-pattern matching and fails to inspect the web traffic nowadays. In addition, Google proposed a compression method called SDCH [29], which is available primarily in Google's related servers and browsers but has not been widely used by other web sites as shown in our experiments. The usage of [30], which can make decompression-free inspection on the traffic compressed by SDCH is also limited since it cannot be extended to LZ77.

VI. Conclusion

In this paper, we have presented Twins for RegEx matching on compressed traffic. Twins stores states returned by scanning each byte of traffic, so as to skip more bytes than the state-of-the-art approach, while introducing less extra cost. The comparisons of Twins with related works draw a significant improvement in speed with real traffic from Alexa top sites. Actually, Twins almost approaches the upper limit on skipping compressed data. Specifically, Twins achieves 1.5Gbps throughput, which enables the potential of wire-speed matching over compressed traffic matching.

VII. Acknowledgement

This work is supported by the National Key Research and Development Program of China (2017YFB0801703), the NSFC (61702407, 61672425) and the Fundamental Research Funds for the Central Universities.

References

[1] J. Fan, C. Guan, K. Ren, Y. Cui, and C. Qiao, "Spabox: Safeguarding privacy during deep packet inspection at a middlebox," *IEEE/ACM Transactions on Networking*, 2017.

[2] C. Hu, H. Li, Y. Jiang, Y. Cheng, and P. Heegaard, "Deep semantics inspection over big network data at wire speed," *IEEE Network*, vol. 30, no. 1, pp. 18–23, 2016.

[3] A. Bremler-Barr and Y. Koral, "Accelerating multipattern matching on compressed http traffic," *IEEE/ACM Transactions on Networking*, vol. 20, no. 3, pp. 970–983, 2012.

[4] X. Sun, K. Hou, H. Li, and C. Hu, "Towards a fast packet inspection over compressed http traffic," in *IEEE/ACM International Symposium on Quality of Service*, 2017, pp. 1–5.

[5] D. Hogawa, S.-i. Ishida, and H. Nishi, "Hardware parallel decoder of compressed http traffic on service-oriented router," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. WorldComp, 2013, p. 1.

[6] M. Becchi, A. Bremler-Barr, D. Hay, O. Kochba, and Y. Koral, "Accelerating regular expression matching over compressed http," in *INFOCOM, 2015 Proceedings IEEE*. IEEE, 2015, pp. 540–548.

[7] L. P. Deutsch, "rfc 1952: Gzip file format specification version 4.3," https://www.rfc-editor.org/rfc/rfc1952.txt, May 1996.

[8] ——, "rfc 1951: Deflate compressed data format specification version 1.3," https://www.rfc-editor.org/rfc/rfc1951.txt, May 1996.

[9] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[10] J. E. Hopcroft and J. D. Ullman, *Introduction To Automata Theory, Languages, And Computation*. China Machine Press, 2007.

[11] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection over encrypted traffic," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 213–226, 2015.

[12] T. Gupta, H. Fingler, L. Alvisi, and M. Walfish, "Pretzel: Email encryption and provider-supplied functions are compatible," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 169–182.

[13] "Compressed traffic data sets," https://github.com/xiuwencs/depict.

[14] "Alexa top 500 global sites," http://www.alexa.com/topsites/.

[15] "Alexa top china sites," http://www.alexa.cn/siterank/.

[16] "Regular expression processor," http://regex.wustl.edu.

[17] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4. ACM, 2006, pp. 339–350.

[18] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM, 2007, pp. 145–154.

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

[20] A. Bremler-Barr, Y. Koral, and V. Zigdon, "Shift-based pattern matching for compressed web traffic," in *High Performance Switching and Routing (HPSR), 2011 IEEE 12th International Conference on*. IEEE, 2011, pp. 222–229.

[21] S. Wu, U. Manber *et al.*, "A fast algorithm for multi-pattern searching," 1994.

[22] Y. Afek, A. Bremler-Barr, and Y. Koral, "Space efficient deep packet inspection of compressed web traffic," *Computer Communications*, vol. 35, no. 7, pp. 810–819, 2012.

[23] S. T. Klein and D. Shapira, "Pattern matching in huffman encoded texts," in *Data Compression Conference, 2001 Proceedings DCC*. IEEE, 2001, pp. 449–458.

[24] A. Daptardar and D. Shapira, "Adapting the knuth-morris-pratt algorithm for pattern matching in huffman encoded texts," in *Data Compression Conference, 2004 Proceedings DCC*. IEEE, 2004, p. 535.

[25] Y. Sun and M. S. Kim, "Dfa-based regular expression matching on compressed traffic," in *Communications (ICC), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1–5.

[26] T. Kida, M. Takeda, A. Shinohara, and M. Miyazaki, "Multiple pattern matching in lzw compressed text," in *Data Compression Conference, 1998 Proceedings DCC*. IEEE, 1998, pp. 103–112.

[27] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa, "A boyer–moore type algorithm for compressed pattern matching," in *Annual Symposium on Combinatorial Pattern Matching*. Springer, 2000, pp. 181–194.

[28] R. S. Boyer, "A fast string searching algorithm," *Communications of the Acm*, vol. 20, no. 10, pp. 762–772, 1977.

[29] J. Butler, W.-H. Lee, B. McQuade, and K. Mixter, "A proposal for shared dictionary compression over http," *Sep*, vol. 8, p. 17, 2008.

[30] A. Bremler-Barr, S. David, D. Hay, and Y. Koral, "Decompression-free inspection: Dpi for shared dictionary compression over http," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1987–1995.