Computer Networks 168 (2020) 106996

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Efficient regular expression matching over compressed traffic

Xiuwen Sun [a], Hao Li [a,*], Dan Zhao [b,a], Xingxing Lu [a], Zheng Peng [a], Chengchen Hu [a]

[a] Ministry of Education Key Lab for Intelligent Network and Network Security, School of Computer Science and Technology, Xi'an Jiaotong University, China
[b] Xi'an University of Finance and Economics, China

ARTICLE INFO

Article history:
Received 6 May 2019
Revised 15 October 2019
Accepted 13 November 2019
Available online 14 November 2019

Keywords:
Compressed traffic matching
Regular expression matching
Pattern matching
Deep packet inspection

ABSTRACT

Nowadays, regular expression matching becomes a critical component of the network traffic detection applications, which describes the fine-grained signature of traffic. Web services tend to compress their traffic for less data transmission, which is a great challenge for regular expression matching to achieve wire-speed processing. In this paper, we propose *Twins*, an efficient regular expression matching method over compressed traffic, which leverages the returned states encoding in the compression to skip repeated scanning. We also present an evaluation model to elaborate the factors that influence the performance of compressed traffic matching methods. Our evaluations demonstrate that Twins could skip ~ 90% compression data and can achieve 1.2 Gbps throughput with a single CPU core. It gains 2.2–3.0 times performance boost than the state-of-the-art works. With a parallel implementation using multiple CPU cores, the throughput could be up to 10 Gbps.

1. Introduction

Deep Packet Inspection (DPI) technique has been promoted from the simple string matching to semantics-based analysis for better serving the emerging scenarios including network optimization, security, big data analysis, etc [1,2]. Regular expressions (RegExs) can describe higher-level semantics than plain strings. Therefore, RegEx matching becomes vital for realizing a DPI system.

Today's web services tend to compress their contents before transmitting them, so as to reduce the transmission volume and latency. It heavily impacts traditional RegEx matching, *aka, Naive* method, which only works for the raw, i.e., uncompressed content. For example, with the modern compression algorithm, the compressed traffic can expect a 20% compression ratio [3], which means the Naive method has to make a 5 × speed-up for maintaining its original performance.

The compressed traffic matching (CTM) is then proposed, which consists of two independent stages: decompression and matching. Since the decompression takes only 3.5% of the time of running the string matching [3] and its throughput can achieve more than 10 Gbps for an ASIC design [4], the first stage is fast enough and not critical. Therefore, the second stage determines the performance of CTM.

The principle to accelerate the matching stage is to leverage the hidden information of the compressed traffic, *e.g.*, the flags added by the compression algorithm. By having a quick glance at those information, many of the traffic can be skipped without byte-to-byte matching. 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. We find that both can be largely improved for previous CTM works. For example, one of the previous works ARCH [5] only skips about 79% of the traffic that Naive method scanned, while theoretically more than 90% bytes encoded in compression segments can be skipped according to the analysis in Section 5. Besides, the previous works incur large extra cost, so they can barely achieve the high-speed processing. As a result, the state-of-the-art works ARCH and COIN [6] can not achieve 500 Mbps throughput according to our evaluation.

In this paper, we propose *Twins* to tap the potential of skipping more bytes with less extra cost. 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 the method can skip more bytes by just checking the previous scanning results.

In the preliminary version of this paper [7], Twins achieves more than 1.2 Gbps throughput and brings more than 2.6 × performance boost to ARCH. In this paper, we further optimize Twins by generalizing its algorithm, which provides faster and more

* Corresponding author.
E-mail address: hao.li@xjtu.edu.cn (H. Li).

https://doi.org/10.1016/j.comnet.2019.106996
1389-1286/

stable performance over different RegEx sets in the high speed cases. Besides, we implement a parallel version of Twins to enable the wire-speed RegEx matching over compressed traffic. We also elaborate how the methods affect the performance by the evaluation model through multiple experiments.

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, which reveals the design space of CTM.
(2) We present Twins, which can skip about 90% bytes of the decompressed traffic with minor extra overhead.
(3) We build the prototypes of Twins with a single CPU core, which can achieve more than 1.2 Gbps throughput, boosting 2.2–3.0 times on throughput than the previous works. We also implement a parallel version, which can exceed 10 Gbps with 13 CPU cores over the real traffic and RegEx sets.

The reminder of this paper is as follows. We present the design space and propose an evaluation model for CTM methods in Section 2. We present the basic idea, algorithm and example of Twins in Section 3 and also introduce the optimized algorithm in this section. Then, we introduce the implementations of the methods in Section 4. The evaluation and experiment analysis are described in Section 5. Related works are listed in Section 6. Finally, we conclude this paper in Section 7.

2. Design space of compressed traffic matching

2.1. Regular expression matching with finite state automata

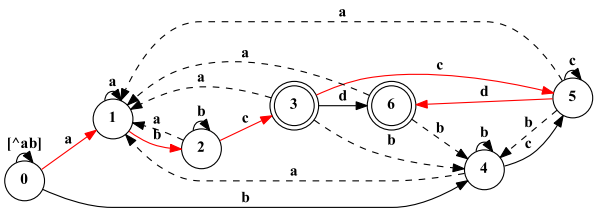
We revisit the de facto structure used in RegEx matching, i.e., the finite state automata (FSA), before we extend it to the context of CTM. The FSA-based method usually compiles RegExs to a nondeterministic finite automaton (NFA) first. Then, the NFA is converted to a corresponding deterministic finite automaton (DFA) and the DFA is minimized. At last, the method finds patterns accepted by this DFA.

In practice, the DFA is organized as a two-dimensional matrix in the memory. Each time the RegEx matching engine reads an input character, and it inquires the matrix once for the next state based on the current state and the input character [8]. Thus, the engine only travels N states when matching an input string with length of N.

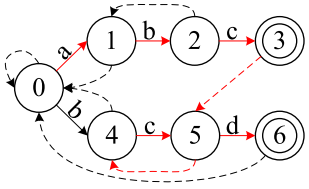
However, the number of states and transitions in the DFA could be explosively growing when converting NFA to DFA, leading to unacceptable memory cost with the rapid growth of patterns in various applications. Therefore, many studies aim to reduce the space consumption of DFA, *e.g.*, D^{2}FA [9], A-DFA [10], namely compression/scalable FSAs [8]. These works can build a much smaller DFA, but at a cost of sub-optimal matching speed in real time.

To have a convenient comparison, we use the same example shown in Fig. 1 which comes from COIN [6]. In the example, Fig. 1(b) shows an A-DFA constructed with the same RegExs "(ab+c)|(bc+d)" in Fig. 1(a). The A-DFA only has 13 transitions rather than 27 in the DFA. The red arrows show the traverse paths of scanning a string "abccd" by the DFA and A-DFA. It is clear that the path "01235456" of the A-DFA is longer than "012356" of the DFA.

While matching the compressed traffic, the simplest CTM method, i.e., Naive method, decompresses it first, and then uses the constructed FSA to scan the decompressed bytes one by one. The decompression stage would enlarge the volume of the data to be scanned, which impacts the performance of matching.



(a) DFA. The solid or dash arrow represents the same meaning, which indicates a transition of eliminating an input character. We omit transitions leading to state 0 for convenience.



(b) A-DFA. Each solid arrow indicates a transition of eliminating an input character. The dash arrow represents the default transition which eliminates nothing. All the transitions have been displayed.

Fig. 1. The examples of DFA and A-DFA, which are constructed with the same RegExs "$(ab+c) \mid (bc+d)$". The state 0 is the start state and double-circle states are the accepting state.

2.2. Design principle of CTM

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

Specifically, more than 90% (434/460) of the Alexa Top 500 sites [11] use *gzip* [12] as their default compression encoding format. *gzip* uses DEFLATE [13] as its compression method, which is a combination of the LZ77 [14] algorithm and Huffman coding. During the compression, LZ77 tries to find repeated maximum occurrences substring with references to the earlier data by employing a sliding window. Next, LZ77 replaces them by a two-tuple of $<$ *length*, *distance* $>$, where the *length* is length of the substring and the *distance* is distance between the substring and the corresponding reference. To unify the terms, the two-tuple is called *pointer* and the corresponding substring is called *referred string*. The positional relation of the terms is shown in the first line of Fig. 3.

After that, the compressed data, which contains pointers and literals (namely the raw content), is usually encoded by dynamic Huffman codes. So, the encoding data is continuous bit stream with variable length and cannot be split in units of bytes. That is why the works of CTM have to decode the traffic before pattern matching. As mentioned, the decompression which contains Huffman decoding and LZ77 decompressing is not critical, so we do not concern it in this paper.

Undoubtedly, the literals have to be scanned and the bytes represented by pointers are redundant in the raw content, which means, if properly used, the previous scanned results can help the downstream matching on the pointers, i.e., the decompressed traffic can be skipped. In our survey, we find the ratio of bytes represented by pointers, which is named ***pointer ratio*** (R_p), is very high ($\geq$ 90%). Now, we show the calculation of the two mentioned ratios with an example in Fig. 3, which ignores the influence of Huff-

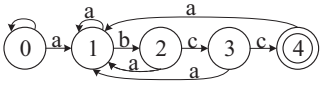


Fig. 2. DFA constructed with string "abcc" and transitions to state 0 are omitted.

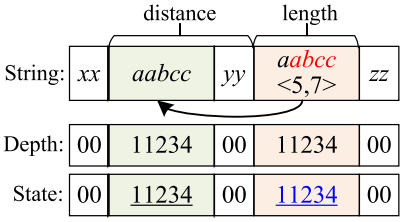


Fig. 3. The example of inspected string with corresponding stored hidden information for string pattern "abcc". The substring "aabcc" between 'y' and 'z' is encoded as $< 5, 7 >$ which represents a pointer, and the bytes not in the pointer are literals. The shadow areas represent the referred string and the pointer respectively. Depth line represents the parameter Depth. State line stores the returned states of scanning the bytes above them.

man encoding and assumes that the pointer takes 3 bytes. Then, the uncompressed traffic size is 16B and the compressed traffic size is 14B by encoding the second substring "aabcc" to " $< 5, 7 >$ ". Thus, the *compression ratio* is 14/16 and the pointer ratio is 5/16, because only 5 bytes are represented by the pointer.

Considering another two examples with Fig. 3, which try to match a string pattern "abcc" with the DFA shown in Fig. 2. Firstly, when Naive method processes the inspected string in the first line, it scans the literals "xxaabccyy" first and meets the pointer later. Then, it gets the original string "aabcc" from the referred string and continues to scan all the bytes. After scanning the last two bytes "zz", it finishes the processing.

Secondly, we assume a simple CTM method (may not be correct) that uses a parameter Depth as the hidden information to accelerate scanning the pointer bytes. The Depth is the shortest length from the current active state to the root state of DFA and could represent the length of string pattern's prefix. During the processing, the method records the Depth for each scanned byte. When it meets a pointer, it copies Depths from the position of the corresponding referred string to the position of the pointer, and gets the Depth of the last byte in the pointer, i.e., 4. Then, it goes backwards 4 bytes, starts scanning from the second 'a' and would match a pattern "abcc".

Comparing with Naive method, this simple method skips the scanning of the first 'a' in the pointer. Here we reveal the potential of saving time for matching one byte, and also incur some extra cost, i.e., recording and copying the stored Depth. Therefore, the design goal of CTM is to skip as many redundant bytes as possible, while bringing the least extra cost.

2.3. Evaluating the performance of CTM

Based on the analysis of CTM, we formalize the evaluation model now. Formally, we use t_s to denote the time consumption of scanning one byte by FSA. Therefore, 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 time of extra cost 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 to the decompressed traffic. Obviously, $R_s = 0$ for Naive method and it can be used as the baseline method. The maximum skipped ratio of the other CTM methods equals to the pointer ratio, which means the scanning of all the bytes represented by the pointers is skipped. As a result, the throughput of CTM can be calculated as follow.

$$T_c = \frac{D}{t_s(1-R_s)D/R_c + t_c R_s D/R_c} = \frac{R_c}{(1-R_s)t_s + R_s t_c}, \quad 0 < R_c \le 1, 0 \le R_s < 1 \tag{1}$$

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

For the scalable FSAs, their parameters t_s are larger than the one of conventional DFA. Therefore, it is difficult to satisfy the requirement of high-speed matching for a CTM method based on these scalable FSAs.

2.4. Evaluating the previous 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 we can treat it as a specific CTM method. Since $R_s = 0, t_c = 0$, its throughput can be calculated as $T_n = R_c/t_s$ according to the Eq. (1). Compared with the throughput of uncompressed matching (T_u), 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 over Naive method can be described as Eq. (2). So, the promotion is determined by R_s and t_c/t_s, namely the skipped ratio and the extra cost of skipping one byte to the time consumption of matching this byte.

$$P = \frac{T_c}{T_n} = \frac{t_s}{(1-R_s)t_s + R_s t_c} = \frac{1}{1-(1-t_c/t_s)R_s}, \quad 0 \le R_s < 1 \tag{2}$$

Suppose $t_c = 0$, we would 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. There is a similar result when $t_c \ll t_s$ and the promotion is only determined by R_s in this case. So, only a bit larger skipped ratio would provide a high potential for improving the throughput.

A method would get a significant performance boost with a larger R_s and a smaller t_c/t_s. It may approach high-speed matching only when t_s is small enough either. We will show the influence of performance boost with different t_s of RegEx matching engines in Section 5.

The previous work ARCH uses some parameters including the category of FSA state, *Input-Depth* and *status* as its hidden information. The Input-Depth is similar to the Depth above and can represent the length of RegEx pattern's prefix. The status is a flag for each scanned byte, which marks the byte with *Uncheck, Check, Match*. ARCH classifies the FSA states into two categories (Simple and Complex) during constructing the FSA. When scanning a byte, it calculates Input-Depth according to the categories and updates status for the scanned byte by Input-Depth. After that, when processing a pointer, it uses the status of the bytes in the corresponding referred string to determine whether the pointer bytes could be skipped and to locate the position for restarting new scanning.

Since the content in the pointer and the corresponding referred string are the same, there is no need to recheck the pattern within the pointer, if it has been matched in the referred string. However, ARCH can skip scanning partial bytes in the pointer only when the referred string does not contain any pattern. If not, ARCH has

to scan these bytes of the pattern in the pointer again. So, it remains redundant scanning when the referred string contains any complete pattern and only skips about 79% bytes of the decompressed traffic (Alexa.com Top 500 sites in Dec. 2011) [5], which is not enough for the 91% pointer ratio. Furthermore, its overhead on calculating its hidden information is so high that the throughput would be slowed down.

COIN divides the FSA states into four categories, *i.e.Initial, Begin, End, Normal*, and uses the category of state and the sequence number of Begin/End state as its hidden information. It skips scanning by checking the categories and locates a complete pattern by a pair of Begin/End state. Whether the pointer contains any complete pattern or not, COIN can skip scanning some bytes in the pointer. COIN can skip 85% bytes and achieve more than 20% improvement over ARCH [6]. However, it also incurs large extra cost while finding the complete patterns.

Therefore, the throughput of ARCH and COIN is not enough to meet the requirement of high-speed. In the next section, we propose Twins to pursue the two goals to meet this requirement.

3. Design of Twins

3.1. 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 an input byte.

Therefore, Twins attempts to skip multiple bytes by only checking once, 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 corresponding referred string are identical, and (2) most byte scanning will mismatch the pattern, leading to the same returned states of the FSM, i.e., the start state.

Based on the first observation, we can skip the rest bytes of the pointer string, as long as we find an identical state returned by the same-position byte in the pointer and referred string. This is because, starting at a certain state, DFA behaves exactly the same when scanning the same string.

The second observation ensures that we can find such an identical returned state in the pointer and referred string in the first few bytes. To be specific, we simply assume the mismatching for each byte is an independent event with a probability of 0.6. This is a conservative assumption, since our experiments on the real traces and patterns suggest the mismatching rate for each byte can reach 0.76. When scanning 1 byte in the pointer and referred string respectively, the probability that both of them are mismatching is 0.6^2. Thus, the probability that not finding identical state for this byte is $1-0.6^2$. As a result, after scanning 6 bytes, the probability that no identical state is found for same-position bytes is only $(1-0.6^2)^6 = 0.069$. Considering the average pointer length could reach 15–20 as shown in Table 1, we can expect a performance boost by skipping most bytes in the pointer.

Therefore, we can make a hypothesis of locality principle for CTM that *in most cases, the returned states are the same for the referred strings and their pointers*. And, there lies a chance that using returned states as the hidden information, which does not need to modify the construction of FSA.

For example, in Fig. 3, when DFA begins to scan the bytes in the pointer, the current active state is '0' which is the same as the stored one before its referred string. It will return the same states ("11234") of scanning the bytes ("aabcc") in the referred string and pointer. Thus, the scanning of bytes in this pointer can be replaced by copying the states ("11234" with underline) from the referred string to the pointer and the scanning of bytes followed the pointer can be continued with the last state ('4') in the referred string or pointer.

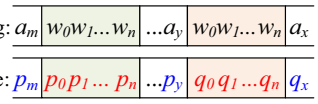


Fig. 4. Inspected string and stored states of Twins.

3.2. Twins algorithm

The name of Twins comes from the feature of the locality principle: most characteristics of the pointer and referred strings are the same, like a pair of twin babies.

The correctness of Twins can be easily ensured, because the next state of FSA is determined by current active state and the next input character, without the influence of previous scanned string, as formally described in Theorem 1.

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

According to the theorem, when processing the bytes $w_0, w_1, \ldots, w_n, a_x$ shown in Fig. 4, Twins checks whether there is any identical returned state in the same-position between p_y, q_0, q_1, ... , q_n and p_m, p_0, p_1, ... , p_n first. Then, it scans the next byte only when the current state is not the same as the stored one before. Without loss of generality, we assume $p_y \neq p_m$ and get $q_0 = p_0$ after scanning the first byte w_0 in Fig. 4. So, $q_x = \hat{\delta}(q_0, w_1, \ldots, w_n a_x) = \hat{\delta}(p_0, w_1, \ldots, w_n a_x) = \delta(p_n, a_x)$. Twins could continue the scanning with p_n, and the bytes $w_1, \ldots, w_n$ in the pointer would be skipped without re-scanning.

The detail of Twins is shown as Algorithm 1 and the function *CompressedMatching* shows the CTM routine on inspecting the literals and pointer bytes of traffic. It invokes FSA procedure to check each literal and keeps the states of scanning them.

When processing the bytes in a pointer by *TwinsScan* procedure, Twins continues to scan them until finding the current state equals to the stored one at the same offset in its referred string. At this time, it stops scanning and copies the states behind this position from the referred string to the pointer. If there are any accepting states, Twins records the state and position as matching result. At last, it uses the last state (q_n in Fig. 4) as the current active state and continues to scan the following bytes behind the pointer.

We will elaborate the algorithm with an example in the next subsection.

3.3. Examples

We use the compressed data in Fig. 5 as input traffic and "$(ab+c) \mid (bc+d)$" as RegEx pattern (the DFA form is shown in Fig. 1(a)) to present the details of Twins on matching pointer bytes. There are two pointers in this example and we denote them as *P*1 and *P*2.

Each sub-figure in Fig. 6 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 *P*1 is equal to the one ('0') in front of its corresponding referred string. So, Twins only needs to copy states ("122" with underline) from the referred string to *P*1, and checks the accepting state during the copying.

Algorithm 1: Twins method.

```
   definition: byteList - array of decompressed traffic;
               stateList - array of returned states len - length
               of pointer; dist - distance between referred string
               to pointer q_0 - the start state of FSA
   input     : Trf_1...Trf_n - compressed traffic
   output    : matched patterns
 1 function CompressedMatching(Trf_1...Trf_n)
 2     curState ← q_0, i ← 0;
 3     for k ← 1 to n do
           // scanning literals
 4         if Trf_i is not pointer(len, dist) then
 5             curState=FSAScanByte(i, curState, Trf_k);
 6             stateList[i] = curState;
 7             i++;
           // processing pointers
 8         else
 9             byteList.Add(byteList[i − dist : i − dist + len]);
10             curState=TwinsScan(i, len, dist, curState);
11             //curState=OptTwinsScan(i, len, dist, curState);
12             i = i + len;
13 function TwinsScan(i, len, dist, state)
14     offset ← (i − dist − 1);
15     for pos in [0, len) do
           // found same states at same offset
16         if state == stateList[offset+pos] then
17             for k ← pos to len do
18                 stateList[i + k] = stateList[offset+k + 1];
19                 if FSA.accept(stateList[i + k]) then
20                     Record stateList[i + k] and i + k;
21             return stateList[i + len − 1];
           // scanning bytes in pointer
22         else
23             state=FSAScanByte(i, state, byteList[i + pos]);
24             stateList[i + pos] = state;
       // have scanned the whole pointer bytes
25     return state;
26 function FSAScanByte(i, state, symbol)
27     state = FSA.lookup(state, symbol);
       // storing matching result
28     if FSA.accept(state) then
29         Record state and i;
30     return state;
```

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

Fig. 5. Input data of example. There are two colored < length, distance > pairs in the compressed data to represent the pointers.

There is no pattern in *P*1, Twins returns the last state ('2') and skips scanning 3 bytes ("abb") in *P*1.

Ex.2: The state ('1') at the position in front of *P*2 is not equal to the one ('2') before its referred string. Twins would have to scan the bytes in *P*2 and return a state ('1') of scanning the first byte, which is the same as the stored state at the same offset in the referred string. So, it copies the following states ("23" with underline) from the referred string to *P*2 and finds a matched pattern "abc" by checking the copied states. Then, Twins returns the state

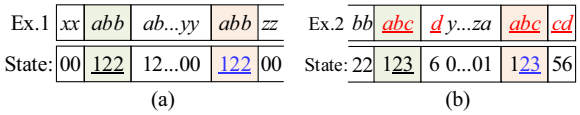


Fig. 6. Examples of matching process of Twins. State line represents returned states of scanning the bytes above them. The bytes with underline are matched patterns and states with underline are copied from the referred string to the pointer.

'3' and continues to scan the bytes behind *P*2. After that, the pattern "bccd" would be found while scanning the following bytes. Twins skips scanning 2 bytes in *P*2.

3.4. Configurable pre-scanning optimization

We can find a conditional branch in the loop of Algorithm 1 between line 15 and 24. It may impact the performance of matching with a fast implementation of DFA. Because, the overhead of a bad branch prediction is high on deeply pipeline architectures with the modern instruction set. So, trying to reduce the probability of misprediction in the loop would boost the throughput of matching.

To achieve this, we further optimize Twins by leveraging a configurable pre-scanning. It could scan some pointer bytes before the conditional branch, and then compares the current state to the stored one. Taking Fig. 4 as an example again, as elaborated previously, Twins checks p_y and p_m before scanning the pointer bytes, while the optimized one may scan w_0 first, and then checks q_0 and p_0. Obviously, the previous elaborated algorithm can be regarded as a special case of Twins, which scans 0 bytes before the conditional branch.

Algorithm 2 lists the detail of Twins with scanning *N* bytes before the conditional branch. The loop between line 2 and 4 of Algorithm 2 should be expanded to avoid producing a new conditional branch and the value of *N* ranges from 0 to 3.

Algorithm 2: Optimized twins algorithm.

```
 1 function OptTwinsScan(i, len, dist, state)
       // pre-scanning N bytes
 2     for n in [0, N) AND n < len do
 3         state=FSAScanByte(i + n, state, byteList[i + n]);
 4         stateList[i + n] = state;
       // the following is similar as previous
          algorithm
 5     offset ← (i − dist − 1);
 6     for pos in [N, len) do
           // found same states at same offset
 7         if state == stateList[offset+pos] then
 8             for k ← pos to len do
 9                 stateList[i + k] = stateList[offset+k + 1];
10                 if FSA.accept(stateList[i + k]) then
11                     Record stateList[i + k] and i + k;
12             return stateList[i + len − 1];
           // scanning bytes in pointer
13         else
14             state=FSAScanByte(i, state, byteList[i + pos]);
15             stateList[i + pos] = state;
16     return state;
```

Twins would improve the accuracy of prediction by pre-scanning pointer bytes and the more pointer bytes are scanned, the higher accuracy of prediction will be achieved. As long as the cost

of pre-scanning the bytes is smaller than that of mis-prediction in branch, it would bring better performance than that does not pre-scan any of these bytes.

To illustrate the influence of the pre-scanning bytes and mis-prediction on the performance of Twins, we use Twins-N to denote Twins scans N bytes before the conditional branch. Then, we suppose the number of literals and pointers of the input traffic as B_0 and B_1 which are fixed value for a specific traffic. We also suppose the number of bytes scanned in the conditional branch of Twins (line 14 in Algorithm 2 or line 23 in Algorithm 1) as B_2. It is clear that B_2 will be decreased with the increasing of N.

Therefore, the total number of scanned bytes of Twins-N is $B = B_0 + NB_1 + B_2$ and that of Twins-N$'$ is $B' = B_0 + N'B_1 + B_2'$, where N' and B_2' are the number of bytes scanned before or in the conditional branch respectively. Without loss of generality, we assume $N > N'$, then the cost of Twins-N to scan the extra bytes is $C_s = (B - B')t_s = (N - N')B_1t_s + (B_2 - B_2')t_s$, and t_s denotes the time consumption of scanning one byte, which has been mentioned before.

After that, we simply assume t_m as the time wastage of predicting incorrectly once even it should be different in each time. The saved time of Twins-N to reduce the mis-prediction would be $C_m = (B_2' - B_2)t_m$.

At last, we define the ratio of bytes scanned in the conditional branch to the decompressed traffic as *mis-prediction ratio* (R_m), namely, B_2/D or B_2'/D. Now, we can get $C_s - C_m = (N - N')B_1t_s - (B_2' - B_2)(t_s + t_m)$ and divide both sides of the equation by the size of decompressed traffic D. Thus,

$$\frac{C_s - C_m}{D} = \frac{(N - N')B_1t_s - (B_2' - B_2)(t_s + t_m)}{D}$$
$$= (N - N')B_1t_s/D - (R_m' - R_m)(t_s + t_m)$$

In the equation, R_m and R_m' are the mis-prediction ratio of Twins-N and Twins-N$'$ respectively. Obviously, Twins-N would be faster than Twins-N$'$ only if $C_s - C_m < 0$, which means

$$R_m' - R_m > (N - N')\frac{B_1}{D}\frac{t_s}{t_s + t_m}. \quad (3)$$

Otherwise, it is not. The right side of Eq. (3) is determined by $N - N'$, t_s and t_m, because D/B_1 is the average pointer length and B_1/D must be a fixed value for a specific traffic. So, the reduction of R_m is limited, which makes Twins-N difficult to maintain the inequality while $N > 1$. It means Twins can hardly gain better throughput by pre-scanning more than two bytes. We will illustrate that in Section 5.

4. Implementation

In this section, we implement Twins based on different FSA construction. Please note that Twins can also be used for the Thompson's NFA [15] by simulating a set of equivalent DFA. However, our goal is to promote the high-speed CTM, so we do not consider the NFA-based Twins.

As analysed in the evaluation model in Eq. (2), the speedup of CTM depends on t_c/t_s. Hence, we invoke two kinds of FSA construction with different t_s to evaluate the performance boost. Besides, the two types of FSA are suitable for different scenarios by considering the trade-off between matching speed and memory cost.

The first implementation is based on the conventional DFA, which is organized as a two-dimensional matrix. As mentioned before, it only needs a few memory accesses to scan one character and can be regarded as a fast implementation of FSA with a small value of t_s.

The second one is based on A-DFA which is organized as a linked list and is used by COIN and ARCH. Since A-DFA travels more states than DFA does and the elements in its list are not adjacent in memory, it would consume far more than a few memory accesses on scanning one character. We treat it as a slow implementation of FSA with a large t_s.

In sum, we implement two prototypes of Twins, i.e., Twins based on DFA and A-DFA. Meanwhile, we also implement two prototypes of COIN and ARCH based on the two FSAs for the evaluation. The two types of FSAs are also used as the base-line algorithms to exhibit the Naive method.

At last, We use the RegEx processor proposed in [16] to construct the conventional DFA and A-DFA.

5. Evaluation

5.1. Settings

We collect two sets of traffic as the input in our experiments. All the raw traffic data are compressed and collected through browsing home pages of Alexa.com top 500 sites [11] and Alexa.cn top 20,000 sites [17]. They can be accessed in [18] and their characteristics are shown in Table 1. For straightforward comparison with ARCH, we take three RegEx sets, i.e., the Snort24, Snort31 and Snort34, which were taken from Snort, used by ARCH and COIN, published at [16].

Since the CTM methods have to decode the traffic before matching, we evaluate the performance of matching decoded traffic. The throughput of each method is calculated with the size of compressed traffic and the time of matching decoded data. All the experiments are performed on a machine with Xeon E5-2630 v4 CPU (2.2 GHz, 10 cores and 20 threads), 64 GB RAM.

5.2. Performance comparison

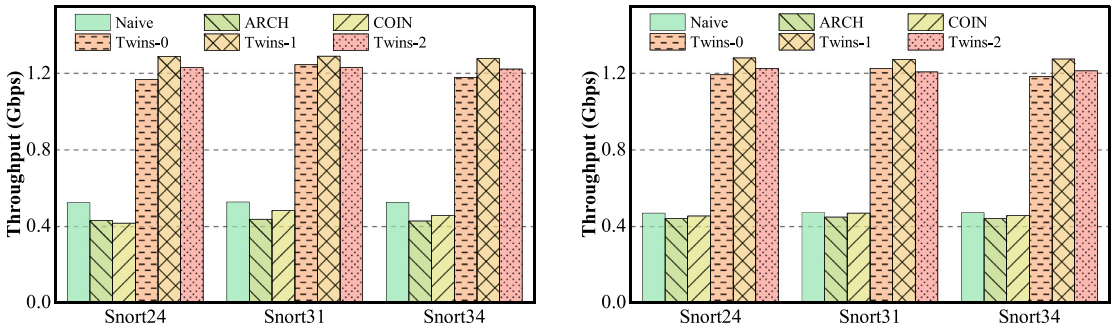
At first, Twins and COIN, ARCH have matched the same number of patterns as Naive method does, which is shown in Table 3. Then, we compare the throughput of Twins and the other methods. They are single-thread programs and run over a single CPU core.

Fig. 7 shows the throughput of implementations based on DFA over the two sets of compressed traffic and three RegEx sets. Each group represents the throughput of the methods over a same RegEx set which is labeled in the horizontal axis. Twins-0 ~ Twins-2 represents the results of Twins on scanning 0 ~ 2 byte(s) before the conditional branch. Obviously, Twins gains more than 2.2 times on throughput than COIN, ARCH and Naive method, over all the data and RegEx sets. Moreover, the throughput of Twins can achieve more than 1.2 Gbps, which means there is potential to perform wire-speed RegEx matching over compressed traffic with a parallel implementation.

Considering the evaluation model in Eq. (2), a smaller extra cost t_c of Twins brings a smaller value of t_c/t_s. It gives Twins better performance than Naive method. Twins achieves 2.2–2.7 times throughput compared to Naive method in Fig. 7. In contrast, both COIN and ARCH get worse results in this situation, because the extra cost of skipping pointer bytes is larger than scanning them, which makes t_c/t_s of COIN or ARCH bigger.

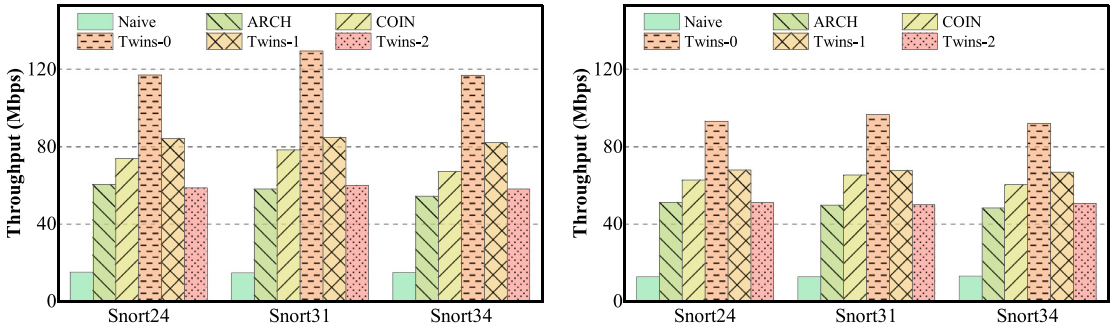
Table 1
Characteristics of experimental data sets.

	Alexa.com	Alexa.cn
Count of Pages	434	13742
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



(a) Alexa.com data set (b) Alexa.cn data set

Fig. 7. Evaluation results based on DFA implementations over two data sets and three RegEx sets.



(a) Alexa.com data set (b) Alexa.cn data set

Fig. 8. Evaluation results based on A-DFA implementations over two data sets and three RegEx sets.

COIN or ARCH may outperform Naive method in some scenarios that the matching engine is implemented by slower FSAs, such as A-DFA, XFA. In these cases, t_s would be larger than t_c, but the throughput of matching would be reduced.

As evidence, Fig. 8 shows the throughput of implementations based on A-DFA. In this case, ARCH can achieve 4.0 times throughput compared to Naive method and COIN can achieve 5.2 times to do that. But Twins-0 can gain 9.2 times throughput compared to Naive method. Furthermore, it is 1.91–2.27 times to ARCH and 1.48–2.22 times to COIN. However, their throughput cannot meet the high-speed in this situation.

At last, we evaluate the memory overhead with the three RegEx sets. The results are shown in Table 2. There is little difference of Twins on the memory consumption, when it scans various number of bytes. Hence, we just invoke Twins-0 in the comparison. From the table, we can find that Twins incurs a little extra memory compared to Naive method and the extra memory is smaller than that of COIN or ARCH. Compared to the memory consumption of Naive method, the extra memory incurred by Twins is negligible. Moreover, it is a fixed value and is independent of the traffic and most matching engines, which will be discussed in Section 5.5.

Table 2
Memory size of the matching engines based on DFA and A-DFA over the three RegEx sets (KB).

	RegEx Set	Naive	ARCH	COIN	Twins
DFA	Snort24	11,360	11,380	11,396	11,360
	Snort31	6872	6892	6928	6872
	Snort34	12,676	12,688	12,708	12,676
A-DFA	Snort24	2252	2268	2280	2256
	Snort31	1648	1664	1692	1652
	Snort34	2540	2552	2560	2544

As a result from the above experiments, not only can Twins achieve better performance, it also saves the extra memory. We will analyse the causes of the higher throughput in the next.

5.3. Analysis

5.3.1. Comparison of Twins with COIN and ARCH

For quantitatively analyzing the reason for the throughput promotion, we compare Twins, COIN and ARCH with another metric, *skipped ratio* (R_s). Table 3 shows the skipped ratio of them on pro-

Table 3
The number of matched patterns and the skipped ratio (R_s) of the compared methods over the two data and three RegEx sets (%).

Data Set	RegEx Set	Matched patterns	ARCH	COIN	Twins-0	Twins-1	Twins-2
Alexa.com	Snort24	20,781	78.67	82.94	**89.91**	84.98	78.94
	Snort31	0	78.41	84.30	**90.82**	85.05	78.95
	Snort34	259	75.96	81.06	**89.70**	84.54	78.51
Alexa.cn	Snort24	357,907	82.42	85.77	**91.28**	87.23	82.65
	Snort31	0	82.52	87.08	**91.81**	87.28	82.66
	Snort34	18,059	81.19	85.37	**91.22**	87.06	82.48

Table 4
Mis-prediction ratio (R_m) of Twins over the two data and three RegEx sets (%).

Data Set	RegEx Set	Twins-0	Twins-1	Twins-2
Alexa.com	Snort24	1.31	0.11	0.02
	Snort31	0.40	0.04	0.01
	Snort34	1.51	0.54	0.44
Alexa.cn	Snort24	0.65	0.07	0.01
	Snort31	0.12	0.01	0.01
	Snort34	0.71	0.23	0.19

cessing the traffic with different RegEx sets. It is clear that Twins-0 skips more bytes than COIN or ARCH does in all the sets. Particularly, Twins-0 skips about 90% bytes on both data sets and almost approaches the theoretical upper bound which can be calculated by the pointer ratio of the two data sets (91.21% and 91.92%, shown in Table 1).

Comparing the skipped ratio in Table 3 and the throughput in Fig. 7, Twins-1 and Twins-2 gain significant higher throughput than that of COIN and ARCH, even their skipped ratios are similar to that of COIN and ARCH. So, we can find that COIN or ARCH incurs more extra cost than that of Twins on skipping the same pointer bytes. 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. COIN locates the complete patterns occurred in the pointers by finding the paired Begin/End states, which also costs more time. Therefore, the more bytes are skipped and the lower extra cost is incurred, the higher performance will be achieved. That is exactly demonstrated by the evaluation model.

As to the results in Table 3 and Fig. 8, both of the throughput and pointer ratio of Twins-1 and Twins-2 are similar to that of COIN and ARCH. It is ascribed to the case of $t_c \ll t_s$ and the promotion of throughput is mainly affected by the pointer ratio (R_s), which is deduced by Eq. (2).

5.3.2. Influence of pre-scanning optimization

Now, we will analyse the influence of pre-scanning bytes on Twins. In Fig. 7, Twins-1 shows better performance than Twins-0. But it is not sustained for the comparison of Twins-2 to Twins-1. Moreover, there are exact opposite results between Twins-1 and Twins-0 in Fig. 8.

To illustrate the reason, we calculate the mis-prediction ratios (R_m) of Twins which are shown in Table 4. It is easy to find that less than 2% traffic bytes are scanned while Twins processes pointers, which proves the correctness of our hypothesis of locality principle in CTM.

From the table, we can find the reductions of mis-prediction from Twins-1 to Twins-2 (less than 0.1%) are smaller than the ones of Twins-0 to Twins-1 (0.11% ~ 1.2%). When $N - N^{'} = 1$, $R_m^{'} - R_m \leq 0.1\%$ is hardly to confirm the inequality of Eq. (3). So, Twins-1 could perform better performance than Twins-0, but Twins-2 could not achieve that than Twins-1. Particularly, it needs more reduction of R_m to hold the inequality of Eq. (3) with $N - N^{'} > 1$. So, the throughput of Twins-2 is not as good as that of Twins-0 in the cases of Snort31 in Fig. 7.

In addition, Twins-0 and Twins-1 bring the opposite results between Figs. 7 and 8. The main reason can be ascribed to the larger t_s of the implementation of A-DFA. With a small t_s of Eq. (3) in Fig. 7, $\frac{t_s}{t_s+t_m}$ would be smaller than 1, which ensures Twins-1 better performance than Twins-0. But with a large t_s in Fig. 8, $\frac{t_s}{t_s+t_m} \approx 1$, Twins-1 could not reduce the mis-prediction ratio enough to hold the inequality of Eq. (3), even it eliminates all the scanning in the pointers. It has to spend more time on scanning the extra byte in each pointer, which leads to the insignificance of saving the time by improving the probability of accurate prediction.

In another way, the pre-scanning optimization can reduce the standard deviation of the throughput among the three RegEx sets over a specific data set, which can be found in Table 5. So, Twins would provide more stable performance over the same traffic and different RegEx sets, when it pre-scans some pointer bytes.

In practice, we can consider the system requirements and estimate the characteristics of traffic and RegEx set to determine which implementation should be employed. When implemented by a fast FSA, Twins can pre-scan one byte. Otherwise, it does nothing before comparing the states.

5.4. Parallel implementation

Twins achieves more than 1.2 Gbps throughput with a single core. It is possible it can exceed 10 Gbps with multiple cores, which could satisfy the high-speed requirements of most DPI systems. To confirm it, we implement the parallel prototypes of DFA-based Twins (only Twins-0 and Twins-1) by using one core to assign the traffic and the other cores to inspect part of the traffic independently.

As shown in Fig. 9, we can find that all the results of different traffic and RegEx sets exceed 10 Gbps with 13 cores. Besides, the result lines of the same traffic set in Fig. 9(b) are close to each other. That proves, once again, the performance of Twins is more stable over different RegEx sets while it pre-scans some bytes.

5.5. Discussion

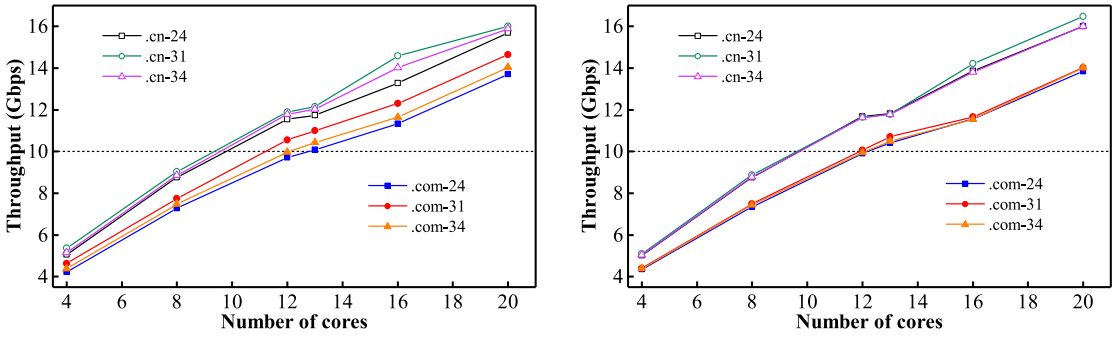
5.5.1. Estimation of extra memory consumption

Twins accelerates the speed of matching by keeping returned states as the hidden information. It only needs 32K-entries to store them, because the maximum distance between the pointer and its referred string is 32,768 B, which is specified by DEFLATE. It is enough to store states using 32K $\times$ 4B memory, which would present more than 4 billion states for DFA. So, the requirement of memory space is invariant for most DFA matching engines.

Compared with hundreds or thousands of million-bytes memory consumption of a DFA engine, thousands-bytes extra cost for Twins is fairly insignificant. Moreover, ARCH or COIN also needs extra space to keep some parameters in their algorithms and FSA states, such as the category of state, Input-Depth and status. Because of the influence of memory allocation algorithm, the comparison in Table 2 does not precisely reflect the difference in memory usage among Twins and the other methods. But from the analysis above, we can learn that it is a common situation that Twins incurs less extra overhead than ARCH or COIN on the extra memory consumption.

Table 5
Standard deviation of throughput of Twins based on DFA and A-DFA.

Data Set	DFA			A-DFA		
	Twins-0	Twins-1	Twins-2	Twins-0	Twins-1	Twins-2
Alexa.com	41.16	7.58	5.07	7.22	1.41	0.92
Alexa.cn	22.49	3.26	9.04	2.38	0.56	0.57



(a) Twins-0 (b) Twins-1

Fig. 9. Throughput of parallel implementations based on DFA. Each line represents Twins performing the CTM over one traffic and one RegEx set.

5.5.2. Estimation of extra time consumption

From Algorithm 1 and the example above, we can learn that Twins utilizes copying states instead of scanning these bytes again. To be simplified, it needs only 2 memory accesses which is smaller than the consumption of scanning one input character. This is why Twins could obtain better performance than the previous works.

Moreover, if all the states in the pointer can be copied at the same time, Twins can achieve better performance. For example in line 18 of Algorithm 1, Twins copies states one by one. If there is no overlap between the pointer and its referred string, this process can be optimized by the *memcpy* function, i.e., copying states in units of blocks.

5.5.3. Matching encrypted traffic

Today's web traffic is often encrypted to protect the data confidentiality and integrity. There are some works [19,20] that focus on DPI over encrypted traffic. Twins does not concern the matching of encrypted traffic. But it can be embedded in the systems to process the decrypted traffic.

5.5.4. Optimization of delay

In this paper, we only use throughput as the major performance metrics of CTM. When deploying Twins to a full DPI system in practice, the delay should be considered as well, because the software-based DPI system costs much more time (may up to several milliseconds) on processing the received packets. In contrast, the smart network interface card (Smart-NIC) can perform pattern matching directly on the specific hardware, such as FPGA, preventing the processing delay from the system kernel.

Our preliminary work shows that Twins on FPGA can achieve a much lower delay, i.e., 15 ns with 200 MHz clock frequency. However, the FPGA-based approaches is faced with the challenge of the restricted resource to implement the FSA and to copy states from a referred string to a pointer efficiently. We leave this valuable work in the future.

6. Related work

6.1. Regular expression matching

The survey [8] concludes RegEx matching for DPI from applications, algorithms and hardware platforms. Considering FSA and scalable FSA, Thompson provides an algorithm for regular expression search and a method to translate regular expressions into NFAs in [15]. The subset construction, which converts any NFA into an equivalent DFA, has been well described in the book [21]. Besides D^2FA and A-DFA, there are also many other works on compressing DFA states or transitions to solve the inflation problem of DFA, such as [22–26]. There are also existing parallel RegEx matching solutions on various platforms including [27–30]. For the applications, Snort [31] is an open source network intrusion detection and prevention system, which employs string and RegEx matching. Its rule sets are widely used in academia and industry. None of these literatures concern how to accelerate the compressed traffic matching.

6.2. With gzip/DEFLATE

ACCH [3] is based on the Aho-Corasick algorithm [32] for compressed traffic matching and skips matching partial bytes when the referred string does not contain any pattern. However, if not, it has to scan these bytes again. SPC [33] employs the same basic idea of ACCH to accelerate multi-string matching over compressed traffic for Wu-Manber algorithm [34]. SOP [35] was proposed to reduce the memory usage of ACCH. However, its speed is relatively lower than ACCH, because it does not provide any optimization on cutting the redundant matching, but incurs more overhead on re-compressing the traffic.

All the methods above are concerned with the acceleration of string matching. Besides, ARCH [5] employs the same basic idea of ACCH and updates the calculation of a parameter to provide RegEx matching over compressed traffic. COIN [6,36] eliminates the redundant scanning of ACCH and ARCH while processing the complete patterns occurred in the pointers. It gets better performance than them. Sun [37] presents another method 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 and it has been discussed in ARCH.

6.3. With other compression methods

The paper [38] provides multi-pattern matching in LZW compressed data and the other two papers [39,40] only apply single-pattern matching to Huffman-encoded data. They are not suitable for the inspection over compressed HTTP traffic which only supports LZ77 compression algorithm. The paper [41] applies Boyer-Moore (BM) algorithm [42] to compressed traffic for fast matching. However, it fails to perform RegEx matching, because the BM is just a single-string matching algorithm. In addition, Google has proposed a compression method SDCH [43], which is built upon the VCDIFF [44] compression data format and available primarily in Google's related services, but has not been widely used by other web sites as shown in our experiments. The usage of [45], which can make decompression-free inspection over the traffic compressed by SDCH, is also limited since it cannot be extended to *gzip* format.

7. Conclusion

In this paper, we have presented Twins for accelerating RegEx matching over compressed traffic. Twins stores returned states of scanning each byte of traffic, so as to skip more bytes than the state-of-the-art approaches, while incurring less extra cost as well. We also propose a model to analyse the effect of the speed of different RegEx matching engines on the performance of compressed traffic matching methods.

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 bound on skipping scanning the compressed traffic. Specifically, Twins achieves more than 1.2 Gbps throughput over single core and exceeds 10 Gbps of parallel implementations, which enables the potential of wire-speed matching over compressed traffic.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

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

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.comnet.2019.106996.

References

[1] J. Fan, C. Guan, K. Ren, Y. Cui, C. Qiao, Spabox: safeguarding privacy during deep packet inspection at a middlebox, IEEE/ACM Trans. Netw. 25 (6) (2017) 3753–3766.
[2] C. Hu, H. Li, Y. Jiang, Y. Cheng, P. Heegaard, Deep semantics inspection over big network data at wire speed, IEEE Netw 30 (1) (2016) 18–23.
[3] A. Bremler-Barr, Y. Koral, Accelerating multipattern matching on compressed HTTP traffic, IEEE/ACM Trans. Netw. 20 (3) (2012) 970–983.
[4] D. Hogawa, S.-i. Ishida, 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, WorldComp, 2013, pp. 1–7.
[5] M. Becchi, A. Bremler-Barr, D. Hay, O. Kochba, Y. Koral, Accelerating regular expression matching over compressed HTTP, in: Proceedings of the 2015 IEEE Conference on Computer Communications, IEEE, 2015, pp. 540–548.
[6] X. Sun, H. Li, D. Zhao, X. Lu, Z. Peng, C. Hu, Coin: a fast packet inspection method over compressed traffic, J. Netw. Comput. Appl. 127 (2019) 122–134.
[7] X. Sun, H. Li, X. Lu, D. Zhao, Z. Peng, C. Hu, Towards a fast regular expression matching method over compressed traffic, in: Proceedings of the 2018 IEEE/ACM International Symposium on Quality of Service, IEEE, 2018, pp. 1–6.
[8] C. Xu, S. Chen, J. Su, S. Yiu, L.C. Hui, A survey on regular expression matching for deep packet inspection: applications, algorithms, and hardware platforms, IEEE Commun. Surv. Tutor. 18 (4) (2016) 2991–3029.
[9] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, J. Turner, Algorithms to accelerate multiple regular expressions matching for deep packet inspection, ACM SIGCOMM Comput. Commun. Rev. 36 (4) (2006) 339–350.
[10] M. Becchi, P. Crowley, A-DFA: a time-and space-efficient DFA compression algorithm for fast regular expression evaluation, ACM Trans. Archit. Code Optim. 10 (1) (2013) 4.
[11] Alexa.com, alexa top 500 global sites, 2017, http://www.alexa.com/topsites/, accessed May.
[12] L.P. Deutsch, Gzip file format specification version 4.3, 1996a, https://www.rfc-editor.org/rfc/rfc1952.txt, accessed Oct. 2019.
[13] L.P. Deutsch, Deflate compressed data format specification version 1.3, 1996b, https://www.rfc-editor.org/rfc/rfc1951.txt, accessed Oct. 2019.
[14] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory 23 (3) (1977) 337–343.
[15] K. Thompson, Programming techniques: regular expression search algorithm, Commun. ACM 11 (6) (1968) 419–422.
[16] M. Becchi, Regular expression processor, 2016, http://regex.wustl.edu, accessed Dec.
[17] A. Alexa top china sites, 2017, http://www.alexa.cn/siterank/, accessed Feb.
[18] X. Sun, Compressed traffic data sets, 2018, https://github.com/xiuwencs/depict, accessed Dec.
[19] J. Sherry, C. Lan, R.A. Popa, S. Ratnasamy, Blindbox: deep packet inspection over encrypted traffic, ACM SIGCOMM Comput. Commun. Rev. 45 (4) (2015) 213–226.
[20] T. Gupta, H. Fingler, L. Alvisi, 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.
[21] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, third ed., China Machine Press, 2007.
[22] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, A.D. Pietro, An improved DFA for fast regular expression matching, ACM SIGCOMM Comput. Commun. Rev. 38 (5) (2008) 29–40.
[23] R. Smith, C. Estan, S. Jha, XFA: faster signature matching with extended automata, in: Proceedings of the 2008 IEEE Symposium on Security and Privacy, IEEE, 2008, pp. 187–201.
[24] R. Antonello, S. Fernandes, D. Sadok, J. Kelner, G. Szabó, Design and optimizations for efficient regular expression matching in DPI systems, Comput. Commun. 61 (2015) 103–120.
[25] X. Yu, L. Bo, M. Becchi, Revisiting state blow-up: automatically building augmented-fa while preserving functional equivalence, IEEE J. Sel. Areas Commun. 32 (10) (2014) 1822–1833.
[26] X. Yu, W. Feng, D. Yao, M. Becchi, O3FA: a scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection, in: Proceedings of 2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ACM, 2016, pp. 1–11.
[27] X. Yu, M. Becchi, GPU acceleration of regular expression matching for large datasets: exploring the implementation space, in: Proceedings of the ACM International Conference on Computing Frontiers, ACM, New York, NY, USA, 2013, pp. 18:1–18:10.
[28] Z. Zhao, X. Shen, On-the-fly principled speculation for FSM parallelization, in: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, 2015, pp. 619–630.
[29] Y. Fang, T.T. Hoang, M. Becchi, A.A. Chien, Fast support for unstructured data processing: the unified automata processor, in: Proceedings of the 48th International Symposium on Microarchitecture, ACM, New York, NY, USA, 2015, pp. 533–545.
[30] X. Yu, K. Hou, H. Wang, W. Feng, Robotomata: A framework for approximate pattern matching of big data on an automata processor, in: 2017 IEEE International Conference on Big Data, IEEE, 2017, pp. 283–292.
[31] Cisco, snort, 2019, https://www.snort.org/, accessed Oct.
[32] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340.
[33] A. Bremler-Barr, Y. Koral, V. Zigdon, Shift-based pattern matching for compressed web traffic, in: Proceedings of the IEEE 12th International Conference on High Performance Switching and Routing, IEEE, 2011, pp. 222–229.
[34] W. Sun, M. Udi, A fast algorithm for multi-pattern searching, University of Arizona, 1994 Tech. rep. tr-94-17.
[35] Y. Afek, A. Bremler-Barr, Y. Koral, Space efficient deep packet inspection of compressed web traffic, Comput. Commun. 35 (7) (2012) 810–819.
[36] X. Sun, K. Hou, H. Li, C. Hu, Towards a fast packet inspection over compressed HTTP traffic, in: Proceedings of the 2017 IEEE/ACM International Symposium on Quality of Service, IEEE, 2017, pp. 1–5.
[37] Y. Sun, M.S. Kim, DFA-based regular expression matching on compressed traffic, in: Proceedings of the 2011 IEEE International Conference on Communications, IEEE, 2011, pp. 1–5.
[38] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, Multiple pattern matching in lzw compressed text, in: Proceedings of the Data Compression Conference, IEEE, 1998, pp. 103–112.
[39] S.T. Klein, D. Shapira, Pattern matching in Huffman encoded texts, in: Proceedings of the Data Compression Conference, IEEE, 2001, pp. 449–458.
[40] D. Shapira, A. Daptardar, Adapting the Knuth–Morris–Pratt algorithm for pattern matching in Huffman encoded texts, Inf. Process. Manag. 42 (2) (2006) 429–439.
[41] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, S. Arikawa, A Boyer–Moore type algorithm for compressed pattern matching, in: Annual Symposium on Combinatorial Pattern Matching, Springer, 2000, pp. 181–194.
[42] R.S. Boyer, A fast string searching algorithm, Commun. ACM 20 (10) (1977) 762–772.
[43] J. Butler, W.-H. Lee, B. McQuade, K. Mixter, A proposal for shared dictionary compression over HTTP, 2008, https://lists.w3.org/Archives/Public/ietf-http-wg/2008JulSep/att-0441/Shared_Dictionary_Compression_over_HTTP.pdf, accessed Feb. 2017.
[44] D. Korn, J. MacDonald, J. Mogul, K. Vo, The vcdiff generic differencing and compression data format, https://www.rfc-editor.org/rfc/rfc3284.txt, accessed Oct. 2019 2002.
[45] A. Bremler-Barr, S. David, D. Hay, Y. Koral, Decompression-free inspection: DPI for shared dictionary compression over HTTP, in: Proceedings of the 2012 IEEE Conference on Computer Communications, IEEE, 2012, pp. 1987–1995.

Xiuwen Sun received his Ph.D. degree in computer science from Xi'an Jiaotong University in 2019 and is now an assistant professor in the School of Computer Science and Technology at Anhui University. His research interests are network measurement and network security.

Hao Li received his Ph.D. degree in computer science from Xi'an Jiaotong University in 2016 and is now an assistant professor in the School of Computer Science and Technology at the same university. His research interests are network measurement and software defined networking.

Dan Zhao is a Ph.D. student in the School of Computer Science and Technology at Xi'an Jiaotong University and works in Xi'an University of Finance and Economics. Her research interests are network measurement and network security.

Xingxing Lu received his M.S. degree in computer science from Xi'an Jiaotong University. His research interest is software defined networking.

Zheng Peng received his B.S. degree in computer science from Xi'an Jiaotong University in 2013 and is now a Ph.D. student in the School of Computer Science and Technology at Xi'an Jiaotong University. His research interests are software defined networking and network measurement.

Chengchen Hu is a Principal Engineer at Xilinx Inc. and is the director of Xilinx Labs Asia Pacific(XLAP), which he founded in Aug. 2017. Prior to joining Xilinx, he was a Professor and the Department Head at the Department of Computer Science and Technology, Xi'an Jiaotong University in P. R. China. He is recipient of the New Century Excellent Talents in University award from Ministry of Education, China, a fellowship from the European Research Consortium for Informatics and Mathematics (ERCIM), a fellowship of Microsoft "Star-Track" Young Faculty. This work was mainly done when he was with Xi'an Jiaotong University.