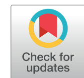
Journal of Network and Computer Applications 127 (2019) 122–134

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

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

COIN: A fast packet inspection method over compressed traffic



Xiuwen Sun [a], Hao Li [a,*], Dan Zhao [a,c], Xingxing Lu [a], Kaiyu Hou [b], Chengchen Hu [a]

[a] Ministry of Education Key Lab for Intelligent Network and Network Security, Department of Computer Science and Technology, Xi'an Jiaotong University, China
[b] Department of Electrical Engineering and Computer Science, Northwestern University, USA
[c] Xi'an University of Finance and Economics, China

ARTICLE INFO

Keywords:
Multi-pattern matching
Regular expression matching
Compressed traffic
Deep packet inspection

ABSTRACT

Matching multiple patterns simultaneously is a key technique in Deep Packet Inspection systems, such as firewall, Intrusion Detection Systems, *etc*. However, most web services nowadays tend to compress their traffic for less data transferring and better user experience, which has challenged the original multi-pattern matching method that work on raw content only. The straightforward solutions directly match decompressed data which multiply the data to be matched. The state-of-the-art works skip scanning some data in compressed segments, but still exist the redundant checking, which are not efficient enough. In this paper, we propose COmpression INspection (COIN) method for multi-pattern matching over compressed traffic. COIN does not recheck the patterns within compressed segment if it has been matched before, so as to further improve the performance of matching, we have collected real traffic data from Alexa top sites and performed the experiments. The evaluation results show that COIN achieves 20.3% and 17.0% in the average of improvement than the state-of-the-art approaches on the string and regular expression matching with real traffic and rule sets.

1. Introduction

Deep Packet Inspection (DPI) technique has been promoted as a critical component in many scenarios from the traditional security ones including firewalls and Intrusion Detection System (IDS) (Levandoski et al., 2008; SourceFire, 2017) to the emerging ones, such as network optimization, big data analysis (Hu et al., 2016), *etc*. The tasks become challenging because of the significant trend that the web service traffic today is transmitted after compression. Specifically, 66% of Alexa top 1000 sites used HTTP compression in July 2010 (Afek et al., 2012) and the ratio of the top 500 sites has increased to more than 90% in May 2017 (Alexa.com, 2017).

To address the challenge of pattern matching in DPI over compressed traffic, the traditional method, namely Naive, matches the decompression traffic byte-by-byte, which is quite slow because of the multiplied data to be matched after decompressing.

To save the matching time, the state-of-the-art ACCH (Bremler-Barr and Koral, 2012) reuses the information obtained from the decompression phase to accelerate the string matching over compressed traffic. The other state-of-the-art ARCH (Becchi et al., 2015), which can accelerate regular expression (RegEx) matching over compressed traffic, leverages the same algorithm as ACCH but replaces calculation of a parameter. However, we observe that they also incur redundancy in matching phase, leading to the inefficient processing.

We observe there exists redundancy when directly rechecking the decompressed content, because these bytes are identical with the plain texts that must have been checked before. By removing such redundancy, in the preliminary version of this paper (Sun et al., 2017), we propose a novel COmpression INspection (COIN) method over the compressed traffic, which outperforms ACCH by 20.3% averagely under the real traffic data. In this paper, we further propose an algorithm for accelerating RegEx matching over compressed traffic, which is 17% faster than the state-of-the-art in average. To be specific, we make the following contributions:

- We present a more efficient method COIN, which contains two algorithms to accelerate string and RegEx matching over compressed traffic respectively.
- We implement the prototype of COIN which achieves a better performance improvement on throughput of matching than the state-of-the-art works.
- We prove the correctness of the two algorithms.

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

https://doi.org/10.1016/j.jnca.2018.12.008
Received 21 May 2018; Received in revised form 14 October 2018; Accepted 4 December 2018
Available online 7 December 2018
1084-8045/© 2018 Elsevier Ltd. All rights reserved.

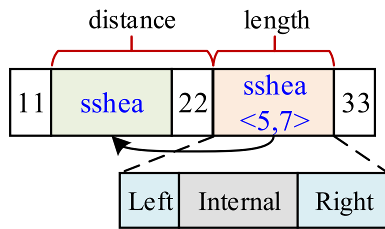


Fig. 1. Illustration of the pointer and its referred string. The left shadow area represents a referred string and the right one represents a pointer.

The reminder of this paper is as follows. Section 2 briefly reviews the background of compression inspection and the technical problems with ACCH and ARCH. Section 3 presents the design of COIN, including the details of two algorithms and examples. Section 4 describes the experiments and our evaluation results. Related works are presented in Section 5, and in Section 6, we conclude the paper. Finally, we list the proofs in appendix.

2. Background and problem statement

2.1. Background knowledge on gzip

Gzip (Deutsch, 1996b) is a compression encoding format recommended by HTTP 1.1 and is utilized by more than 90% (434/460) of the web sites as its default format shown in our survey on Alexa Top 500 sites. Gzip uses DEFLATE (Deutsch, 1996a) as its compression method, which is based on the combination of the LZ77 (Ziv and Lempel, 1977) algorithm and Huffman coding.

At first, LZ77 maintains a sliding window as a dynamic dictionary for compression. While scanning the plaintext, LZ77 will try to find the maximum substring in the window, and record its length and distance between the current position and the substring in the window.

Fig. 1 shows gzip compression addressing the example text “11sshea22sshea33”. LZ77 compresses the second plaintext “sshea” to a two-tuple of “<length, distance>” where the length is the length of the plaintext and the distance is the offset from the tuple to the plaintext, namely “<5, 7>” in the case. That means the original text can be restored by copying 5 bytes from the offset position of 7 bytes. To unify the terms, the first plaintext “sshea” is called *referred string* and “<5, 7>” is named as *pointer*.

After that, the compression data, which contains literals and pointers, is encoded by Huffman coding. Therefore, the encoding data is continuous bit stream with variable length that cannot be byte-coded. It is also the reason why the works have to decompress traffic before a pattern matching method applied.

At last, the encoding data is saved as DEFLATE format and added gizp header and tail, which includes magic number, CRC result, *etc*. In the specification of DEFLATE, the size of distance of pointer is [1, 32768] and the range of length is [3,258]. Using the length, we define two indicators, the *average pointer length* which is easy to understand and the *pointer ratio* as the ratio of bytes represented by pointer to the decompressed traffic. They determine the potential of improvement on inspecting compression traffic which will be discussed in section 4.

2.2. Multi-pattern matching and challenge on compressed traffic

In general, multi-pattern matching over compressed traffic consists of two independent stages: decompression and matching. The first stage is fast enough and not critical (Hogawa et al., 2013), so the second stage determines the performance of the whole process.

In this paper, we focus on Aho-Corasick (AC) (Aho and Corasick, 1975) algorithm for string matching and finite state automata (FSA) for RegEx matching. AC constructs a DFA-like FSA by patterns first, and then adds *failure function* for each state to allow searching any other prefix branch while current searching failed. It processes the input characters in a single pass with deterministic performance.

For example, Fig. 2 shows a FSA of AC for recognizing the string patterns “her”, “she” and “abc”. The red arrows indicate the path while scanning the string “sher”. In the scanning process, each transition with solid arrow eliminates one character. But, transitions with dash arrow eliminate nothing.

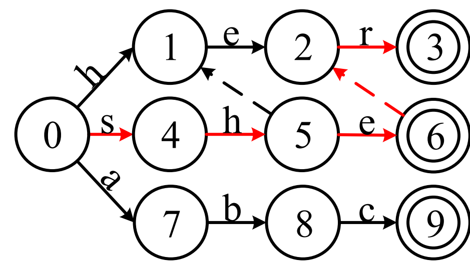


Fig. 2. AC FSA of string patterns ‘her’, ‘she’, ‘abc’. Each solid arrow indicates a transition of scanning a character. Each dash arrow represents a failure function. The other failure functions to root state are omitted.

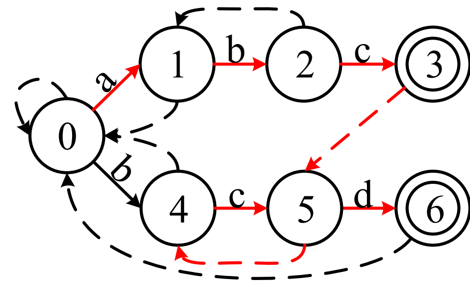


Fig. 3. A-DFA for patterns ‘ab + c’, ‘bc + d’. Each solid arrow indicates a transition of scanning a character. The dash arrows represent the default transitions.

The algorithms based on FSA are widely used for RegEx matching. It usually compiles RegExs to a nondeterministic finite automaton (NFA) first. Then, it converts the NFA to a corresponding deterministic finite automaton (DFA) and minimizes the DFA. At last, it finds strings which is accepted by the DFA.

The converting of NFA to DFA leads to the explosion of DFA states and transitions, which makes DFA costs so much memory space than NFA. There are many studies to reduce the space consumption of DFA with some matching performance losing, such as, D^2FA (Kumar et al., 2006), A-DFA (Becchi and Crowley, 2007), XFA (Smith et al., 2008).

For example, Fig. 3 shows a A-DFA constructed with the same RegExs in Fig. 6. It only has 13 transitions rather than 27 in Fig. 6. The *default transition* of A-DFA is similar to the failure function of AC. The red arrows indicate the path “01235456”, when the A-DFA scans a string “abccd”.

Either failure function or default transition, travels more than N states when processing a string of length N. These algorithms are called compression or scalable FSAs in the survey (Xu et al., 2016) and cost much more time, but less memory space than regular DFA does on matching characters. For example, XFA achieves 108 Mbps throughput with small memory sizes in its evaluation.

Moreover, with amounts of compressed traffic and low compression ratio (the average ratio is 20% in our survey), the matching over compressed traffic becomes a challenge that makes the throughput of matching of Naive over compressed traffic less than one fifth of that over uncompressed traffic.

2.3. Previous works

The characters of the pointer are the same to the referred string's. If methods could skip some characters in pointer, they would accelerate the matching process. ACCH and ARCH are the methods for string and RegEx matching respectively.

ACCH is fundamentally a multi-string matching approach based on AC algorithm. The basic observation in ACCH is: "If referred string does not completely contain matched patterns then the pointer contains none". Following this observation, pointer is separated to three parts, *Left, Internal*, and *Right* (shown as Fig. 1).

In the Left case, the pattern starts prior to the pointer and its suffix is in the pointer. ACCH uses a parameter *depth*, namely, the shortest length of a simple path from the current scanned state to the root state of AC, to determine whether a pattern ends within the left boundary of pointer. If the current state depth, prior pointer area is 0, ACCH moves directly to the Internal case without scanning a single byte. If not, it continues scanning the pointer bytes as long as the number of scanned bytes within the pointer is smaller than the depth.

In cases of Internal and Right, ACCH maintains a parameter *status* and marks bytes with the flag of *Uncheck, Check, Match* during the scanning. A byte with Uncheck means its depth is smaller than a pre-defined constant parameter *CDepth* (2 as default), otherwise the byte is marked Check. And a byte is marked with Match while any patterns end at this position.

The Right case represents that the pointer contains a pattern prefix and its remaining bytes occur after. ACCH locates the last occurrence of Uncheck within the referred string and scans from the corresponding position *unchkPos* within the pointer. In order to determine safely the right boundary, the scan is resumed from $unchkPos - CDepth + 2$.

The Internal case represents that a pattern ends within the referred string. Same as the Right case, ACCH starts scanning at the position of $unchkPos - CDepth + 2$ and this unchkPos is the last occurrence of Uncheck prior the byte with Match. If none of the matched patterns occurred within the referred string, ACCH would skip the Internal area of the pointer. However, if not, it has to scan those bytes again. For example, we assume that the string "she" is a pattern which locates in the Internal area of the pointer in Fig. 1 and ACCH has to scan these bytes even they have been found in referred string before.

Due to the closure ($*$ or $+$), the depth could not represent the length of pattern's prefix any more in RegEx matching. For example, the prefix length of a string "abcc" at last byte is 4. But, the depth is 3 in the DFA shown in Fig. 3. So, ACCH can not perform its algorithm in this scenario directly.

ARCH is the state-of-the-art work based on the same idea and algorithm of ACCH to perform RegEx matching over compressed traffic. The key effort behind ARCH is to calculate a parameter *Input-Depth* for each scanned character which replaces the depth in ACCH.

Input-Depth depends on the automaton state and the scanned bytes. It is the length of the shortest suffix of scanned string in which inspection starting at start state (q_0) and ending at current state (s). For DFA, ARCH uses two methods, *simple and complex states* and *positive and negative transitions*, to calculate the upper bound as the value of Input-Depth.

In *simple and complex states* method, ARCH marks the states with simple or complex flag during the DFA construction procedure. When it travels a simple state, it sets the value of Input-Depth with the depth of DFA, which is similar to the one in ACCH. And it increases the value by one when traveling a complex state. In *positive and negative transitions* method, ARCH defines two types of transitions: a positive transition which increases the Input-Depth by one and a negative-transition which either decreases the value or leaves the value unchanged. In the next sections, we will only introduce the *Simple/Complex* method since it is the only one evaluated by ARCH.

After calculated Input-Depth, ARCH uses the same procedures of ACCH to perform matching. Thus, it is also insufficient for losing sight of redundant scanning in the case of complete pattern in pointer. Furthermore, the overhead of calculating the Input-Depth is so large that impacting the performance improvement.

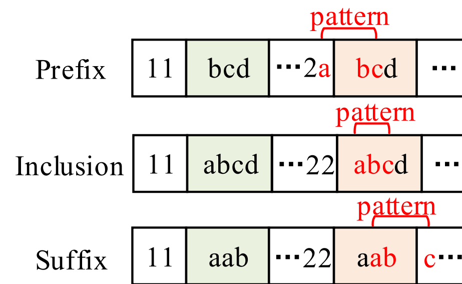


Fig. 4. Categories of COIN. The string 'abc' is a pattern to be matched.

3. Design of COIN

In this section, we propose a faster method COIN for multi-pattern matching over compressed traffic. Since the contents of the pointer and referred string are identical, we get another observation that if there is a complete pattern in the referred string, then the pattern must be also in the pointer.

Based on this observation, COIN inspects the uncompressed data after decompression and stores parameters for accelerating matching. It leverages different ways to match patterns according three cases shown in Fig. 4.

(1) *Prefix*, pattern starts before pointer, but regardless of its end position.
(2) *Inclusion*, pattern is contained in pointer entirely.
(3) *Suffix*, pattern starts in pointer, but not contained.

Obviously, the three cases are basic and complete and the Proof is listed in Appendix A. They are only the positional relationships between the pointer and pattern and can be assembled to any other complex cases, such as Fig. 12(b) which combines two cases: Suffix (*pointer1*) and Prefix (*pointer2*).

COIN uses a different case classification and different processing method as what ACCH does in Inclusion and Suffix cases. The basic idea of COIN is to confirm whether a complete pattern occurred in the pointer and find the last appropriate position for restarting matching.

Algorithm 1 shows the routine of COIN on inspecting literals and the pointer data of compressed traffic. It invokes FSA procedure to check each literals and processes bytes in pointer with different methods for string and RegEx matching scenarios. We will introduce them in the following subsections.

3.1. String matching

As mentioned in ACCH, the depth could represent the length of pattern's prefix. It can be also used to process the case of Prefix and Suffix in COIN. With the addition of length of matched pattern, It can be determined that whether a pattern within pointer completely. So, COIN handles the three cases as following and the detail is shown in Algorithm 2. The Proof of correctness is listed in Appendix B.

(1) *Prefix*: COIN uses the same parameter depth defined in ACCH. It ends this procedure directly and moves to process Inclusion, if depth is equal to 0 at the position in front of the pointer. Otherwise, it would not stop scanning the pointer bytes until the number of scanned bytes is smaller than the depth.

(2) *Inclusion*: COIN introduces a parameter called *status*, which indicates whether a pattern is matched at that location. If patterns are matched during the matching process, COIN records their length and locations as matching information and marks the locations with *m (Match)*. When handling Inclusion case, COIN copies status from the

Algorithm 1 COIN (COmpression INspection).

```
definition : byteInfo - {byte, depth, status, cate, etc}
             byteList - array of byteInfo
input      : Trf_1...Trf_n - compressed traffic
1  function COIN(Trf_1...Trf_n)
2      curState ← q_0;
3      for i ← 1 to n do
4          if Trf_i is not pointer(len, dist) then
               //scan literals in traffic
5              byteInfo.byte = Trf_i;
6              curState=FSAScanByte(byteInfo, i, curState);
7              byteList.Add(byteInfo);
8          else
               //scan pointer by DEPICT method
9              byteList.Add(byteList[i − dist : i − dist + len]);
10             curState=StringScan(i, len, dist, curState);
11             // curState=RegExScan(i, len, dist, curState);

12 function FSAScanByte(byteInfo, index, state)
13     byteInfo.state = FSA.lookup(state, byteInfo.byte);
14     if FSA.accept(byteInfo.state) then
15         Record byteInfo.state and index;
16     Set parametres of byteInfo;
17     return byteInfo.state;
```

referred string to the pointer first. Then let *mPos* be the position in the pointer with Match status, and *mLen* be the length of pattern. By checking whether the $mPos - mLen$ is out of the left boundary of pointer, we can detect whether an entire pattern is appeared or not. With the appearance of complete pattern, COIN keeps and stores its length and location, otherwise COIN eliminates the Match status at *mPos*.

(3) *Suffix*: Copying depth to the pointer and determining whether a Suffix case is presented by detecting the depth of the last byte (*lastPos*) in the pointer. If the depth is larger than 0, COIN restarts the scanning from the position at $lastPos - depth$.

3.2. Regular expression matching

For RegEx matching, COIN gets another way to find the last appropriate position for restarting matching because the depth can not represent the length of patterns' prefix. Before elaborating the detail of it, we present four categories to identify the states of FSA. The definitions of them are listed as follows.

Initial State: The states in the $\epsilon - closure$ of start state q_0 in NFA. When using the *Subset Construction* algorithm (Hopcroft and Ullman, 2007) converts NFA to DFA, a NFA subset that all the states of it are marked with Initial states generates an Initial state of DFA. For example, the state marked with green circle is an Initial state in Fig. 6.

Begin State: The states that returned by $\delta(q_0, a)$ and a is the first symbol in a RegEx. In Fig. 6, the states marked with blue circle are Begin states.

End State: The accepting states. To identify the Begin or End states, they are numbered with sequence number of RegExs.

Normal State: All the other states except the three categories above.

In this scenario, COIN stores the category of each state and the sequence number of Begin/End states while scanning each byte of traffic. Then, it restarts the matching by finding the last appropriate position marked with a Begin state instead of finding the position with the depth in string matching. It also checks that whether there are any patterns occurring in pointer completely by finding the pair of Begin/End state with a same sequence number. COIN processes the pointer bytes by three cases which is same as the string matching and the detail is shown in Algorithm 3. The Proof is listed in Appendix C.

(1) *Prefix*: COIN checks the stored state category at the position in front of the pointer. If it is an Initial State that means there is no pattern started before pointer and it is no need to process this case. Otherwise, COIN has to continue the scanning to find the possible pattern until the FSA returns an Initial state.

(2) *Inclusion*: COIN finds the first position marked with a Begin state, which is denoted as *scanPos*, and keeps *curState* as the state for restarting scanning. It uses a set *setBegin* to store the sequence number of all the Begin states found in the pointer. After that, When an End state is found, COIN checks whether its sequence number matches the one in *setBegin*. If it does, COIN records the End state and current position as matching information, moves *scanPos* to the position followed by the current one and changes *curState* to the End state.

(3) *Suffix*: If COIN finds an Initial state or repeated Begin state which is the same as the one at *scanPos*. It moves *scanPos* to the current checked position and changes *curState* only when finding an Initial state. For any other situations, it keeps the position of *scanPos*. At last, COIN restarts a new scanning with *curState* and the byte at *scanPos*.

Algorithm 2 COIN - String Matching.

```
input : i - index of traffic
        length - length of pointer
        dist - distance between referred to pointer
        curState - state of reading prior byte of pointer
1  function StringScan(i, dist, len, curState)
2      curPos ← 0; offset ← i − dist;
       //=== 1. prefix case ===
3      if byteList[i − 1].depth ≠ 0 then
4          for curPos ← 0 to len do
5              k ← i + curPos;
6              curState=FSAScanByte(byteList[k], k, curState);
7              if byteList[k].depth ≤ curPos then
8                  break;
9      if curPos ≥ len − 1 then break;
       //=== 2. inclusion case ===
10     for j ← curPos to len do
11         if byteList[i + j].status = MATCH then
               //pointer contains entire pattern
12             if match[i + j].len ≤ j + 1 then
13                 curPos ← j + 1;
14                 Record match[i + j].len and i + j;
15             else
16                 byteList[i + j].status ← 0;
       //=== 3. suffix case ===
17     lastByte ← byteList[i + len − 1];
18     if curPos < len ∧ lastByte.depth ≠ 0 then
19         curState = q0;
20         offset ← i + len − lastByte.depth;
21         while curPos ≤ lastByte.depth ∧ curPos < len do
22             k ← offset + curPos++;
23             curState=FSAScanByte(byteList[k], k, curState);
```

3.3. Example

We use the compressed data in Table 1 as input to compare the pattern matching process. The first line is the plaintext and the second line represents compression data to be scanned. Now, we will elaborate the string and RegEx matching as follows.

- String Matching

We use string patterns in Fig. 2 for string matching and the values of parameters of COIN and ACCH are shown in Fig. 5.

(1) *Prefix*: COIN shares the same process to scan pointer with ACCH. The depth in front of the pointer is 0, so it does not need to process the case of Prefix and moves to process the Inclusion case directly.

(2) *Inclusion*: COIN copies depth and status from the referred string to the pointer. Then, it finds a byte marked with Match in the pointer, thus *mPos* is 3 (the starting offset begins with 0) and the *mLen* is 3 (‖'abc′‖). Therefore, there is an complete pattern in pointer and COIN records it without scanning it again. It is the only complete pattern in pointer, so COIN moves to process the case of Suffix.

When ACCH processes this case, it gets $mPos = 3$ by finding the position marked with Match in the referred string and gets $unchkPos = 1$ by locating the last occurrence of Uncheck before *mPos*. Then, it resumes the scanning at the position of $unchkPos - CDepth + 2 = 1$, which is the first position marked with blue 'u' in the pointer.

Algorithm 3 COIN - Regular Expression Matching.

```
input      : i - index of traffic
             length - length of pointer
             dist - distance between referred to pointer
             curState - state of reading prior byte of pointer
1  function RegExScan(i, len, dist, curState)
       //=== 1. prefix case ===
2      offset ← (i − dist);
3      for curPos ← 0 to len do
4          if byteList[i + curPos − 1].cate==CATE_INIT then break;
5          curState=FSAScanByte(byteList[i + curPos], i, curState);
6      if curPos ≥ len then return;
7
       //=== 2. inclusion and suffix case ===
8      curState = 0;   setBegin.clear();
9      stateId ← −1,   scanPos ← curPos;
10     for j ← curPos to len do
           //skip bytes with initial state
11         if byteList[offset + j].cate== CATE_INIT then
12             scanPos = j;   stateId = −1;   curState = 0;
           //skip bytes with repeated begin state
13         if byteList[offset + j].cate & CATE_BEGIN then
14             if stateId == −1 then stateId = byteList[offset + j].id;
15             if stateId == byteList[offset + j].id then scanPos = j;
16             setBegin.append(byteList[offset+j].beginNo);
           //check if pointer contains pattern?
17         if byteList[offset + j].cate & CATE_END then
18             if byteList[offset + j].endNo in setBegin then
                   //find complete pattern
19                 scanPos = j + 1;   stateId = −1;
20                 curState = byteList[offset + scanPos].state;
21                 Record curState and i + j;
22
       //=== 3. copy stored information ===
23     for k ← curPos to scanPos do
24         byteList[i + k] = byteList[offset + k];
25
       //=== 4. restart new scanning ===
26     for k ← i + scanPos to len + i do
27         curState=FSAScanByte(byteList[k], k, curState);
28     return curState;
```

Table 1
Input data of example.

plaintext	11aabcdab22**aabcdab**cd33
compressed	11aabcdab22< **7, 9** >cd33

string:	11	aabcdab	22	aabcdab	cd33
depth:	00	1123012	00	1123012	3000
COIN:	00	000m000	00	000m000	m000
ACCH:	uu	uucmuuc	uu	uucmuuc	muuu

Fig. 5. Example of COIN and ACCH (CDepth = 2) for string matching. The first line is the decompressed traffic and the red bytes are the ones would match the pattern 'abc'. The second line represents the depth parameter and the last two lines (COIN and ACCH) represent the status parameter defined by their algorithms ('u', 'c' and 'm' are the flags of 'Uncheck', 'Check' and 'Match'). The greens mean the corresponding bytes would be skipped and the blues are the positions of resuming scanning by ACCH. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

So, the complete pattern "abc" has to be scanned again.

(3) *Suffix*: The depth of the last position is 2, so COIN begins a new scanning at the position of $lastPos - depth$ ('a' at 5). In ACCH, it finds the last position marked with Uncheck within the referred string and gets $unchkPos = 5$ in the pointer. Then, it begins a new scanning at $unchkPos - CDepth + 2 = 5$ which is the second blue 'u'. It is the same position as COIN in this example.

In this example, the scanning of bytes corresponding to the green status are skipped by COIN or ACCH. It is clear to find the difference between them from the number of skipped bytes. When ACCH finds a complete pattern in pointer, it has to scan it again, but COIN does not.

- Regular Expression Matching

For convenience, we use the DFA in Fig. 6 and list the parameters of the DFA for COIN and ARCH in Table 2. The depth is the length of shortest path from current state to the start state and the complex represents a simple or complex state ('0' for simple and '1' for complex). Begin and End are the sequence number of Begin and End states. All these parameters can be obtained by the state. For example, the state '6' is a complex state and its depth is 3. Its Begin and End sequence number are '0' and '2'.

Now, we will elaborate the details of COIN for RegEx matching with Fig. 7.

Table 2
Parameters of the DFA for COIN and ARCH.

State	0	1	2	3	4	5	6
depth	0	1	2	3	1	2	3
complex	0	0	1	1	0	1	1
Begin	0	1	2	0	2	0	0
End	0	0	0	1	0	0	2

string:	11	aabcdab	22	aabcdab	cd33
state:	00	1123612	00	1123612	3600
Input-Depth:	00	1123412	00	1123412	3400
status:	00	uucmmuc	00	uucmmuc	mmuu

Fig. 7. Example of COIN and ARCH for regular expression matching. The first line is the decompressed traffic and the red bytes would match the patterns. The second line lists states of checking the bytes above. Particularly, the blues are the Begin states, the reds are the End states and the greens represent the bytes above them would be skipped. The last two lines list the parameters of Input-Depth and status of ARCH. The values of Input-Depth in blue are calculated by complex state and the others are obtained from Table 2. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

(1) *Prefix*: An Initial state is appeared in front of the pointer and COIN won't process this case.

(2) *Inclusion* and *Suffix*: When checking stored states in referred string, COIN finds the first Begin state and gets $scanPos = 0$. The next state is also a Begin state which is the same as the one at *scanPos*, so COIN moves $scanPos = 1$. The third one is also Begin state but not equals the one at *scanPos*, so it keeps $scanPos = 1$. While finding these three Begin states, COIN puts their sequence number into a set which contains 1 and 2.

After that, there is an End state and its sequence number is 1. So, it is a complete pattern ('abc') for finding a pair of Begin/End state with same sequence number. Then, COIN records this pattern information, lets $curState = 3$ and moves $scanPos = 4$. The next one is also an End state with sequence number 2, COIN records the information of the complete pattern ('bcd') and gets $scanPos = 5$, $curState = 6$.

At this moment, it is a Begin state at *scanPos* with sequence number 1. The next one, which is also the last one, is a Begin state with sequence number 2. So, $scanPos = 5$ will be kept.

After checked all the stored states, COIN copies the states ('11236') before *scanPos* from the referred string to the pointer and restarts a new scanning with *curState* ('6') and the byte at *scanPos* ('a'). Thus,

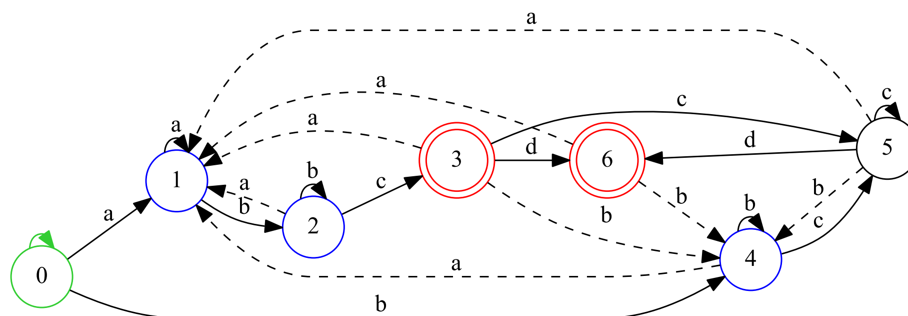


Fig. 6. DFA for '$(ab + c) \mid (bc + d)$'. The green circle state is Initial state, the blues are the Begin state, and double-circle states are the accepting state, *i.e.* End state. We omit transitions leading to state 0 for convenience. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

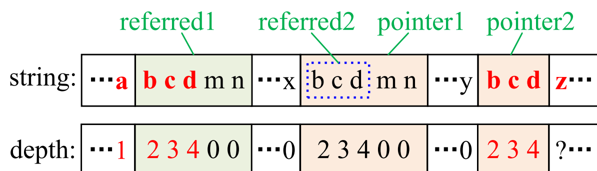


Fig. 8. Influence of multi-referred string on string matching.

Table 3
Characteristics of experimental data sets.

	Alexa.com	Alexa.cn
Number 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

it will find the matched patterns ('abc', 'bcd') after the pointer. The scanning of bytes corresponding to green states are skipped by COIN in this example.

Since the process is similar to the example of ACCH after getting the values of Input-Depth in ARCH, we just elaborate the calculation of the Input-Depth instead of introducing all the details of ARCH. As shown in Fig. 7, when ARCH scans the bytes"aa" in the referred string, the returned states are simple states. Thus, the Input-Depth can be obtained from Table 2 directly. Scanning the next bytes "bcd" would return complex states and these Input-Depth values are incremented by one. Once getting the values of Input-Depth, ARCH would mark the status parameter. Then, it will employ the algorithm of ACCH to find the position for resuming scanning in the pointer.

3.4. Discussion

(1) *Performance and cost*

As the numbers of bytes skipped by ACCH and COIN in the above examples, COIN skips more bytes in the *Inclusion* case. Actually, the analysis at section 4.4 shows that more than 90% of patterns are presented in *Inclusion* case on string matching. Skipping the scanning of bytes in *Inclusion* can bring lots of performance improvements.

The memory requirements of ACCH, ARCH and COIN are similar. Considering string matching, both of COIN and ACCH need memory spaces to store the matching information. The difference of them is COIN needs a high performance for searching the matching results. But it is a little difference in actual memory usage. For the stored parameters on string matching, COIN needs more space to preserve depth parameter, but ACCH has to store it in each state of AC.

For RegEx matching, COIN holds states for scanned bytes and needs some memory to hold category and sequence number in each state of FSA. However, it also needs to keep some information in ARCH algorithm and its FSA states, such as *status, Input-Depth, Simple/Complex state*.

Moreover, only 32 K size of the parameters are needed and preserved for all the methods, because the maximum length of distance between the pointer and referred string is 32768B, which is specified by DEFLATE. Therefore, the requirements of space are scale invariant for a determinate FSA. Compared with hundreds or thousands million bytes memory usage of FSA, thousands bytes extra space is more insignificant.

(2) *Influence of parameters*

The number of skipped bytes in ACCH are not the same with different *CDepth*. For example, if *CDepth* = *1* when ACCH processes the case of *Inclusion* and *Suffix* in Fig. 5, it will have to scan two more bytes, which should be skipped under a well-chosen user-specified parameter. So, ACCH would scan some redundant bytes with an inappropriate parameter.

Fig. 8 shows a example of multi-referred string. The pointer1 and referred2 have some overlapped bytes. Since the scanning process will skip the first three bytes ('bcd') in pointer1, pointer2 cannot get the exact depth from the referred2. Therefore, ACCH has to use *CDepth* to represent the approximate depth.

In *Prefix* case of COIN, it only uses the depth of the byte before the pointer and doesn't use depth in *Inclusion* case. Thus, we only need to discuss the influence of depth in *Suffix* case. Assuming "bcdz" is a pattern which needs to be matched in pointer2, we will lost the pattern only if the depth of last byte ('d') in pointer2 is less than 3. In other words, it occurs only if "bcd" has not been scanned in pointer2 or referred2 and get a smaller depth from referred1.

However, if "bcd" is not a prefix of any other patterns, pointer2 will get a larger depth from referred1. For example, we assume string "abcd" is another pattern. The depth "234" will be copied from referred2 to pointer2 and is larger than the actual value "123". Otherwise, the depth will be its actual value. So, no matter how many references, the depth of final pointer won't be smaller than its actual value, which means, it won't miss any patterns in the case of *Suffix*.

The parameter status of COIN is accurate in multi-referred situation. Assuming "abcd" is a pattern, the third byte ('d') owns a Match status in pointer1. COIN copies status from referred1 to pointer1 first. After finishing processing the *Inclusion* case of pointer1, COIN will update the Match status of the byte to "0" immediately. So, pointer2 will gets an accurate status from referred2.

4. Evaluation

4.1. Implementation

As mentioned, ACCH accelerates pattern matching over compressed traffic based on the AC algorithm and it did not specify the implementation of the algorithm. So, we build AC with a trie and keep depth parameter in each state. After that, we implement the prototypes of ACCH and COIN for string matching.

To have an straightforward comparison on RegEx matching, we implement COIN over RegEx processor at (Becchi, 2016) which contains an implementation of A-DFA used by ARCH. It is also used as the baseline method, *i.e.*Naive. We implement ARCH with the *Simple/Complex* method to calculate the Input-Depth parameter.

4.2. Settings

Before the evaluation, we have collected traffic by accessing the Alexa top sites and upload them into GitHub (Sun, 2018). Their characteristics are shown in Table 3. All the raw traffic data related to the transferring home pages of top 500 Alexa.com (Alexa.com, 2017) (only the list of top 500 sites is available publicly in October, 2016) and top 20000 Alexa.cn (Alexa.cn, 2017) sites using compression, which are used as the input traffic in our experiments.

In addition, we take 1430 string patterns from Snort rules (SourceFire, 2017) for comparison of COIN and ACCH. Due to the same reason for straightforward comparison on RegEx matching, we take three RegEx sets, the Snort24, Snort31 and Snort34, which were taken from Snort, published at (Becchi, 2016) and used by ARCH. Then, we use a desktop PC (Intel i5-4460 and 8G RAM) for evaluation. All implementations are single-threaded programming and run over single core.

4.3. Performance

Firstly, COIN, ACCH and ARCH have matched the same number of patterns as Naive does. Then, we compare their throughput over the

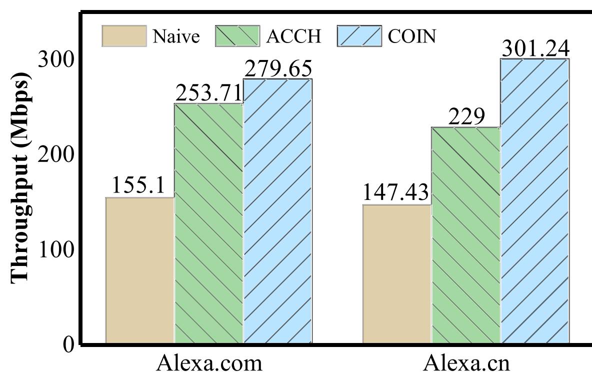


Fig. 9. Throughput comparisons of Naive, ARCH and COIN on string matching over the Alexa.com and.cn sets.

Alexa.com and Alexa.cn data set. Fig. 9 shows the throughput of COIN and ACCH on string matching over the two data sets. COIN achieves an average of 20.3% throughput improvement than ACCH. Fig. 10 shows the throughput of COIN and ARCH over the two data sets and three RegEx sets. It outperforms 17.0% averagely on the throughput than ARCH. Therefore, COIN is more efficient than ACCH and ARCH for string and RegEx matching over both of the data sets.

The next, we compare the memory overhead of the matching engines which are implemented by these methods with the string and RegEx sets. The results show in Table 4. From the table, we find that COIN incurs a little extra memory space than ACCH or ARCH. For example, COIN spends 30076 KB memory space and costs 32 KB more than ACCH for string matching. All the accelerating methods (COIN, ACCH or ARCH) cost about tens KB more space than Naive, which is negligible compared to the memory size of Naive. Moreover, the extra cost is a fixed value for a specific matching engine and is independent of the traffic size.

At last, for quantitative analysing the evaluation results, we compare the methods with another indicator, *skipped ratio* (R_s). Namely, the ratio of skipped bytes of applying the accelerating algorithm to the total size of the decompressed traffic. Obviously, $R_s = 0$ for Naive and the maximum of R_s is equal to the pointer ratio which is represented in Table 3.

Table 5 lists the skipped ratio of COIN, ACCH and ARCH by processing the two data sets with the string and RegEx sets. COIN skips 82.85% bytes of decompressed traffic which have to be scanned by Naive on string matching over the Alexa.cn data set. It also skips 72.84% bytes over the Alexa.com data set. But ACCH only skips 78.17% and 70.02% over the two data sets. It is clear that COIN also skips more bytes than ARCH over both of the data sets and three RegEx sets. So, the more bytes are skipped, the higher performance will be achieved.

Table 4
Memory size of the scanning engines built by the methods with string and RegEx rules (KB).

Rules	Naive	COIN	ARCH	ACCH
Strings	29944	30076	–	30044
Snort24	5912	5968	5944	–
Snort31	5412	5504	5440	–
Snort34	5908	5980	5964	–

Table 5
The skipped ratio over the two data sets and the rules (%).

Data Set	Rules	COIN	ARCH	ACCH
Alexa.com	Strings	72.84	–	70.02
	Snort24	82.94	78.67	–
	Snort31	84.30	78.41	–
	Snort34	81.06	75.96	–
Alexa.cn	Strings	82.85	–	78.17
	Snort24	85.77	82.42	–
	Snort31	87.08	82.52	–
	Snort34	85.37	81.19	–

4.4. Analysis

In order to evaluate the influence of different patterns on the throughput. We compare COIN, ACCH and ARCH with some synthetic rules.

- String Matching

COIN skips the scanning of reoccurring patterns within the pointer to accelerate the string matching. We will evaluate COIN and ACCH with various number of patterns occurred in the Inclusion case while keeping the other factors same. We choose Alexa.com data set as the input traffic and select some groups of occurring patterns from the string pattern set which is used above. The statistics are listed in Table 6 and each group only contains one string pattern. The length of these patterns and the number of total matched patterns are similar in each group. The inclusion matched column lists the number of patterns completely occurring in the Inclusion case.

When none of the patterns have been matched, the main difference on the skipped ratio may be ascribed to the consideration that ACCH determines safely the right boundary by resuming the scanning from the position of $unchkPos - CDepth + 2$. Moreover, the first and the last pattern which have the same prefix “width” produce a higher difference on the boost of the skipped ratio. In general, COIN skips more bytes than ACCH with the increasing of the patterns (the total and matched in the Inclusion case).

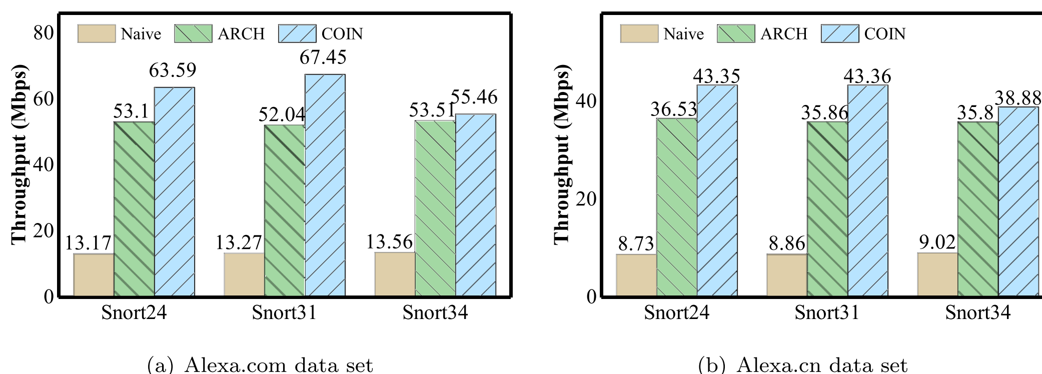


Fig. 10. Throughput comparisons of the methods on RegEx matching over the two data sets (Alexa.com and.cn) and three RegEx sets (Snort24, Snort31 and Snort34).

Table 6
The variation of skipped ratio of COIN and ACCH with synthetic rules over Alexa.com. R_{sc} and R_{sa} are the skipped ratio of COIN and ACCH. $Boost = 100 \times (R_{sc}/R_{sa} - 1)$.

Rules	Total Matched	Inclusion Matched	R_{sc} (%)	R_{sa} (%)	Boost (%)
widthxy	0	–	91.110	90.794	0.348
CDEFGI	0	–	91.176	90.843	0.366
MZXYZ	0	–	91.185	90.853	0.366
LOGIN	41	22	91.195	90.862	0.366
EMF	59	0	91.182	90.849	0.366
MZ	390	149	91.185	90.852	0.367
PK	604	218	91.192	90.859	0.367
host	1865	1434	91.078	90.742	0.370
slug	1919	1838	90.909	90.566	0.380
height	39109	38331	91.038	90.650	0.427
width	47416	46587	91.127	90.756	0.409

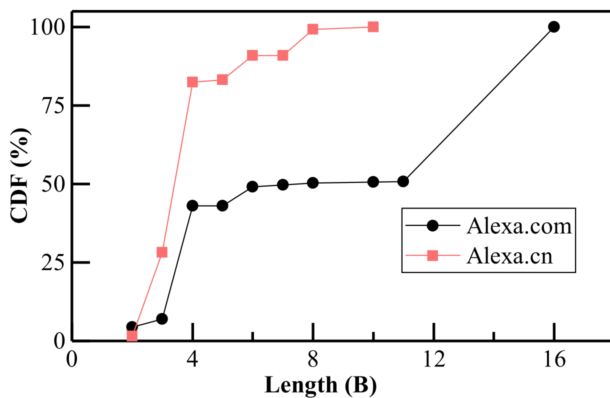


Fig. 11. Cumulative distribution function of the length of matched patterns over the two data sets.

Fig. 11 is the cumulative distribution function of the matched patterns's length of inspecting the Alexa.com and.cn data sets with all the 1430 string patterns. It shows that the length of most matched patterns is smaller than 12 bytes. But the average pointer length of the Alexa.cn and Alexa.com is about 15 bytes, shown in Table 3. So most of the patterns would be occurred in the Inclusion case. Besides, we have checked the matched patterns over the two data sets and confirm that 92.61% matched patterns of Alexa.com and 94.23% of Alexa.cn are entirely contained in the Inclusion case. Therefore, the more patterns are occurred in the Inclusion case, the higher skipped ratio will gain and the better performance will achieve.

- Regular Expression Matching

ARCH may offer a lower performance gain when detecting patterns that contain a short string prefix followed by the closure. Because, these rules would generate more complex states. To illustrate that, we append "*" behind the second characters in each RegEx rules above and denote the new rule sets as Snort24′, Snort31′ and Snort34′. Then, we compare the throughput and skipped ratio of COIN and ARCH over Alexa.com data set. The results show in Table 7.

When we insert "*" into the rule set of the Snort31 and Snort34, the throughput and skipped ratio of COIN has only changed a little (increased or decreased). But, there has been some decrease in ARCH. Especially, when changed the Snort24, the throughput of ARCH decreased 13.71% and the skipped ratio decreased 4.08%. The two measures of COIN changed less than half of ARCH's. Therefore, COIN is not as sensitive as ARCH in this situation.

Table 7
The comparisons of throughput and skipped ratio between COIN and ARCH over Alexa.com. The columns of T and R_s are the throughput and skipped ratio. The Decreased rows are the decrease ratio of T and R_s.

Rule Set	COIN		ARCH	
	T (Mbps)	R_s (%)	T (Mbps)	R_s (%)
Snort24	63.59	82.94	53.10	78.67
Snort24′	60.32	82.13	45.82	75.46
Decreased (%)	**5.14**	**0.98**	**13.71**	**4.08**
Snort31	67.45	84.30	52.04	78.41
Snort31′	67.56	84.32	51.55	77.91
Decreased (%)	**−0.16**	**−0.02**	**0.94**	**0.64**
Snort34	55.46	81.06	53.51	75.96
Snort34′	55.31	81.09	52.80	75.65
Decreased (%)	**0.27**	**−0.03**	**1.33**	**0.41**

4.5. Future work

COIN accelerates multiple pattern matching over compressed traffic by reducing the redundant scanning of complete patterns within the pointer. Finding the complete patterns also incurs some extra cost which limits its throughput for wire-speed matching. In the future, we will study some mechanism to reduce the cost, such as determining the complete patterns in the pointer more quickly or implementing COIN with a hardware platform. And then, we are going to integrate the method into a full DPI system.

5. Related work

5.1. Deep packet inspection

The survey (Xu et al., 2016) has concluded matching methods about DPI from applications, algorithms and hardware platforms. The paper (Bremler-Barr et al., 2014) treats DPI as a service to the middleboxes, implying that traffic should be scanned only once and all middleboxes use the service. The paper (Thompson, 1968) provides an algorithm for regular expression search and introduces a method of compiling RegEx to NFA. To solve the state inflation problem, there are so many researches on DFA state transition compression, such as (Becchi and Crowley, 2007; Kumar et al., 2006; Smith et al., 2008). Snort (SourceFire, 2017) is an open source network intrusion detection and prevention system based on DPI. Its rule sets are widely used in academic and industry. However, all of these literature does not concern how to accelerate the matching over compressed traffic.

5.2. With gzip/DEFLATE

ACCH and ARCH are the methods of accelerating string and RegEx matching over compressed http traffic which have been described in Section 2.3 and thus is omitted here. Besides, there are other works that focus on inspection of the traffic compressed by LZ77. SPC (Bremler-Barr et al., 2011) is another method for accelerating string matching over compressed traffic which is based on Wu-Manber algorithm

(Sun and Udi, 1994) and is similar to ACCH with the same basic idea. SOP algorithm (Afek et al., 2012) 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 (Klein and Shapira, 2001) and (Daptardar and Shapira, 2004) to match on 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, Sun and Kim (2011) presents a method to accelerate RegEx matching over compressed traffic. But, it relies on the compression DFA which have reduced its number of path pairs significantly. That limits its application scenarios.

5.3. With other compression methods

The paper (Kida et al., 1998) achieved multi-pattern matching over the compressed data with LZW, but it cannot work on LZ77 compression algorithm and thus cannot support inspections over HTTP traffic. In (Shibata et al., 2000), the authors applied the Boyer-Moore algorithm (Boyer and Moore, 1977) to compress 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 (Butler et al., 2008), which is available primarily in Google's related servers and browsers but has not been widely used by other web site as shown in our experiments. The usage of (Bremler-Barr et al., 2012), which can make decompression-free inspection over the traffic compressed by SDCH is also limited since it cannot be extended to LZ77.

6. Conclusion

In this paper, we have presented a method called COIN for multi-pattern matching over compressed traffic. Unlike the previous works that check the same pattern each time when it appears in the compressed data segment, COIN only matches the pattern once and skips its future appearance within the compressed segment. It is much faster than the state-of-the-art approaches on the string and RegEx matching with real traffic from Alexa top sites.

Acknowledgement

This work is supported by the National Key Research and Development Program of China (2016YFB0800101), the NSFC (No.61672425, 61402357) and the Fundamental Research Funds for the Central Universities.

Appendix A. Classification completeness of COIN

Theorem 1. *The classification of COIN processing pointer bytes is complete.*

Proof. We prove the completeness through the relative position of pattern and pointer. Let U be a universal set that represents the pattern and pointer intersect. Proving this theorem is equivalent of proving the union of classification is equal to U.

At first, we consider the start position of pattern and divide U to two parts. Let p be the situation that start position of pattern before the pointer. Thus, $\neg p$ means its start position in the pointer and $U = p \vee \neg p$.

Then, we divide $\neg p$ by the end position of pattern. Let q be the situation that the end position in the pointer. So, $\neg q$ means its end position behind the pointer and $\neg p = q \vee \neg q$.

Therefore, It can be derived that $U = p \vee q \vee \neg q$. Obviously, p can be regarded as the Prefix of COIN. q means that the start and end position of pattern are both in pointer. In other words, pointer contains pattern completely, defined as Inclusion of COIN. $\neg q$ represents Suffix of COIN. So, it is proved. □

Appendix B. Correctness of string matching algorithm in COIN

Theorem 2. *COIN detects all the string patterns in compressed traffic as the Naive method does.*

Proof. COIN scans the literals by AC scanner when pattern and pointer are not intersect and scans the pointer bytes according to the classification of COIN when they are intersect. With the Theorem 1, it's easy to verify the completeness of Algorithm 2. So, we prove Theorem 2 through the following four cases.

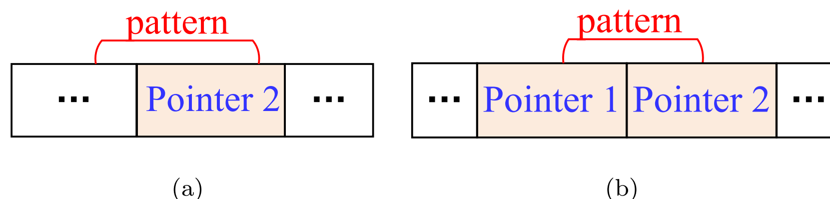


Fig. B.12 The relative position of pattern and Pointer2 belong to the Prefix case.

(1) *Non-intersect:* In this case, all the bytes are literals. COIN scans them by AC which also used by the Naive method. So, they must have the same result.

Before discussing the other three cases, we introduce an auxiliary function σ called suffix function (Cormen et al., 2012). For a given pattern $p[1, \ldots, m]$, the function maps string x *to* $\{0, 1, \ldots, m\}$ such that $\sigma(p, x)$ is the length of the longest prefix of p that is also a suffix of x. Along with the suffix function, we can formally describe the parameter depth as $\max\{\sigma(p_i, x) \mid p_i \in P\}$, where P is the set of patterns.

(2) *Prefix:* As shown in Fig. B.12, the prior byte of Pointer2 may be located in literals or Pointer1. Compared to the Naive method, COIN will get a same depth value in Fig. B.12(a) and get a same or bigger value in Fig. B.12(b) with the discussion (3.4). COIN won't make a wrong decision even the depth is greater than its actual value.

If $depth = 0$ at the prior byte of pointer, it means $\sigma(p, x) = 0, \forall p \in P$. It also means there is no pattern starts before pointer. So, it is not necessary to process this case. Otherwise, COIN will continue scanning the pointer bytes.

The depth will be bigger than the number of scanned bytes in pointer, because the start position of pattern is before pointer. Otherwise, COIN have ended up scanning of current pattern. The remaining bytes do not belong to Prefix. So, COIN could find all the possible patterns in Prefix.

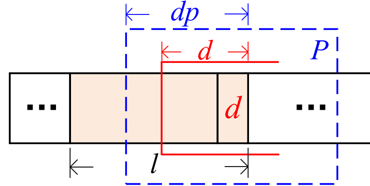


Fig. B.13 Relative position of pattern and the depth in Suffix.

(3) *Inclusion:* It is observed that if there are any complete patterns in the pointer, they must be found from its corresponding referred string. Thus, COIN could find all the string patterns in Inclusion case by estimating whether the length of pattern out of the boundary of the pointer.

(4) *Suffix:* As shown in Fig. B.13, we assume l is the length of pointer and the depth at last position in the pointer is d. At first, We make an opposite assumption that COIN would miss some patterns in this situation. So, it must exist a pattern P and its start position is before the red area, thus $d < dp < l$. According to the definition of the suffix function σ, there is $depth = \max\{d, dp\} = dp$ at the last position in pointer. It derives a contradiction with $depth = d$. Therefore, COIN could find all the patterns in Suffix.

From all the above, it is proved that COIN detects all the string patterns in compressed traffic as the Naive method does. □

Appendix C. Correctness of regular expression matching algorithm in COIN

Theorem 3. *Suppose that* $w[0, 1, \ldots, n]$ *are the bytes of a matched pattern in the referred string and* $p[0, 1, \ldots, n]$ *are the returned states of scanning* w*. As shown in* Fig. C.14*, the states before* p_k *are known and the states* q*, which locate in pointer, are unknown. Thus, there is* $q_x = \hat{\delta}(p_k, wa_x) = \delta(p_n, a_x)$*, where* δ *is the transition function and* $\hat{\delta}$ *is the extended transition function of DFA.*

Proof. We prove it through two cases.

(1) $p_k = p_m$: it is obvious that $\hat{\delta}(p_k, wa_x) = \hat{\delta}(p_m, wa_x) = \delta(p_n, a_x)$.
(2) $p_k \neq p_m$: We have learned that p_n and q_n are accepting states of the same RegEx. So we can get that p_0 and q_0 are equivalent from the definition of equivalence of states. That is means we can always find a way to rename the states so that the two states become the same. It derives that $\delta(p_k, w_0) = q_0 = p_0 = \delta(p_m, w_0)$. Therefore, $q_x = \hat{\delta}(p_k, wa_x) = \hat{\delta}(\delta(p_k, w_0), w_1 \cdots w_n a_x) = \hat{\delta}(\delta(p_m, w_0), w_1 \cdots w_n a_x) = \delta(p_n, a_x)$.

Therefore, it is proved. □

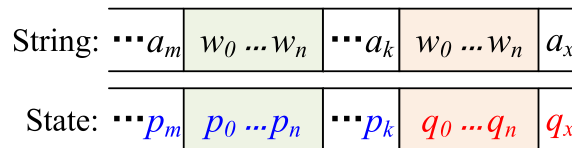


Fig. C.14 Scanned bytes and returned states in the pointer and referred string.

Theorem 4. *COIN detects all the RegEx patterns in compressed traffic as the Naive method does.*

Proof. From the algorithm 2, we can find that COIN detects literals with FSA scanner which is also used in Naive. If the returned state after processing pointer bytes by COIN is equal to the one returned by Naive, COIN must detect all the patterns in compressed traffic as Naive does. So, we can prove the theorem from the three cases of COIN for processing pointer bytes in algorithm 3.>

(1) *Prefix:* If the returned state at prior position before the pointer is an Initial state. It means none of the patterns start before the pointer and is no need to process Prefix case. If not, COIN will detect the following bytes in the pointer until it returns an Initial state. So, it must return an Initial state before processing Inclusion or Suffix cases. Otherwise, it means COIN has detected all the bytes in the pointer and there is no need to process Inclusion and Suffix cases. So, COIN will return the same state as the Naive method does after processing Prefix case.

(2) *Inclusion and Suffix:* As mentioned, there is an Initial state before processing the two cases which is same as Naive. Besides, COIN finds the first position with Begin state as the re-scanning position (*scanPos*). It will return Initial states by scanning bytes before *scanPos*, because the Begin state is the first state by reading the first symbol of any patterns. Therefore, q_0 is the correct state for starting a new scanning if *scanPos* does not change. Then, we will prove that it is safe to move *scanPos* when COIN meets an Initial or repeated Begin state or matched an End state after checking some bytes.

It is obvious that COIN could move *scanPos* to the current position when it meets an Initial or repeated Begin state, because the states are equal at *scanPos* and current position. Through the Theorem 3, we can learn that the scanning of bytes of a complete pattern can be skipped if the pattern

has been matched in its referred string. And COIN also gets a correct state by continue scanning the following bytes with the accepting state of this pattern. So, it could move *scanPos* to the next position of the complete matched pattern which is found by a pair of Begin/End state. Therefore, COIN and Naive also return the same state in these two cases.

From all the above, COIN detects all the RegEx patterns in compressed traffic as the Naive method does. □

References

Afek, Y., Bremler-Barr, A., Koral, Y., 2012. Space efficient deep packet inspection of compressed web traffic. Comput. Commun. 35 (7), 810–819.
Aho, A.V., Corasick, M.J., 1975. Efficient string matching: an aid to bibliographic search. Commun. ACM 18 (6), 333–340.
Alexa.cn, 2017. Alexa Top china Sites. http://www.alexa.cn/siterank/" accessed Feb. 2017.
Alexa.com, 2017. Alexa Top 500 Global Sites. http://www.alexa.com/topsites/" accessed May. 2017.
Becchi, M., 2016. Regular Expression Processor. http://regex.wustl.edu accessed Dec. 2016.
Becchi, M., Bremler-Barr, A., Hay, D., Kochba, O., Koral, Y., 2015. Accelerating regular expression matching over compressed http. In: Proceedings of the 2015 IEEE Conference on Computer Communications (INFOCOM). IEEE, pp. 540–548.
Becchi, M., Crowley, P., 2007. An improved algorithm to accelerate regular expression evaluation. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS). ACM, pp. 145–154.
Boyer, R.S., Moore, J.S., 1977. A fast string searching algorithm. Commun. ACM 20 (10), 762–772.
Bremler-Barr, A., David, S., Hay, D., Koral, Y., 2012. Decompression-free inspection: dpi for shared dictionary compression over http. In: Proceedings of the 2012 IEEE Conference on Computer Communications (INFOCOM). IEEE, pp. 1987–1995.
Bremler-Barr, A., Harchol, Y., Hay, D., Koral, Y., 2014. Deep packet inspection as a service. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT). ACM, pp. 271–282.
Bremler-Barr, A., Koral, Y., 2012. Accelerating multi-pattern matching on compressed http traffic. IEEE/ACM Trans. Netw. 20 (3), 970–983.
Bremler-Barr, A., Koral, Y., Zigdon, V., 2011. Shift-based pattern matching for compressed web traffic. In: Proceedings of the IEEE 12th International Conference on High Performance Switching and Routing (HPSR). IEEE, pp. 222–229.
Butler, J., Lee, W.-H., McQuade, B., Mixter, K., 2008. A Proposal for Shared Dictionary Compression over. https://lists.w3.org/Archives/Public/ietf-http-wg/2008JulSep/att-0441/Shared_Dictionary_Compression_over_HTTP.pdf accessed Feb. 2017.
Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2012. Introduction to Algorithms, third ed. China Machine Press.
Daptardar, A., Shapira, D., 2004. Adapting the knuth-morris-pratt algorithm for pattern matching in huffman encoded texts. In: Proceedings of the Data Compression Conference. IEEE, p. 535.
Deutsch, L.P., 1996. Rfc 1951: Deflate Compressed Data Format Specification Version 1.3. https://www.rfc-editor.org/rfc/rfc1951.txt accessed Feb. 2017.
Deutsch, L.P., 1996. Rfc 1952: Gzip File Format Specification Version 4.3. https://www.rfc-editor.org/rfc/rfc1952.txt accessed Feb. 2017.
Hogawa, D., Ishida, S.-i., Nishi, H., 2013. 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), pp. 1–7.
Hopcroft, J.E., Ullman, J.D., 2007. Introduction to Automata Theory, Languages, and Computation, third ed. China Machine Press.
Hu, C., Li, H., Jiang, Y., Cheng, Y., Heegaard, P., 2016. Deep semantics inspection over big network data at wire speed. IEEE Netw. 30 (1), 18–23.
Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., 1998. Multiple pattern matching in lzw compressed text. In: Proceedings of the Data Compression Conference. IEEE, pp. 103–112.
Klein, S.T., Shapira, D., 2001. Pattern matching in huffman encoded texts. In: Proceedings of the Data Compression Conference. IEEE, pp. 449–458.
Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J., 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. Comput. Commun. Rev. 36 (4), 339–350.
Levandoski, J., Sommer, E., Strait, M., 2008. l7-filter. http://l7-filter.clearos.com/ accessed May. 2017.
Shibata, Y., Matsumoto, T., Takeda, M., Shinohara, A., Arikawa, S., 2000. A boyer–moore type algorithm for compressed pattern matching. In: Annual Symposium on Combinatorial Pattern Matching. Springer, pp. 181–194.
Smith, R., Estan, C., Jha, S., 2008. Xfa:faster signature matching with extended automata. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy. IEEE, pp. 187–201.
SourceFire, 2017. Snort. https://www.snort.org/ accessed Feb. 2017.
Sun, W., Udi, M., 1994. A Fast Algorithm for Multi-pattern Searching. Tech. Rep. TR-94-17. University of Arizona.
Sun, X., 2018. Compressed Traffic Data Sets. https://github.com/xiuwencs/depict accessed May. 2018.
Sun, X., Hou, K., Li, H., Hu, C., 2017. Towards a fast packet inspection over compressed http traffic. In: Proceedings of the 2017 IEEE/ACM International Symposium on Quality of Service. IEEE, pp. 1–5.
Sun, Y., Kim, M.S., 2011. Dfa-based regular expression matching on compressed traffic. In: Proceedings of the 2011 IEEE International Conference on Communications (ICC). IEEE, pp. 1–5.
Thompson, K., 1968. Programming techniques: regular expression search algorithm. Commun. ACM 11 (6), 419–422.
Xu, C., Chen, S., Su, J., Yiu, S., Hui, L.C., 2016. A survey on regular expression matching for deep packet inspection: applications, algorithms, and hardware platforms. IEEE Commun. Surv. Tutor. 18 (4), 2991–3029.
Ziv, J., Lempel, A., 1977. A universal algorithm for sequential data compression. IEEE Trans. Inf. Theor. 23 (3), 337–343.

Xiuwen Sun is a Ph.D. student in the Department of Computer Science and Technology at Xi'an Jiaotong 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 Department 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 Department of Computer Science and Technology at Xi'an Jiaotong University and also 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.

Kaiyu Hou received his M.S. degree in computer science from Xi'an Jiaotong University in 2017 and is now a Ph.D. student in the Department of Electrical Engineering and Computer Science at Northwestern University, USA. His research interests are networking and system.

Chengchen Hu received his Ph.D. degree from Tsinghua University, China, in 2008. He worked in Tsinghua University as an assistant professor (July. 2008–Dec. 2010) and then later severed in Xi'an Jiaotong university as associated professor (Dec. 2010–Jan. 2016) and professor since 2016, all are with the Department of Computer Science and Technology. Since the summer of 2017, he has been on leave and directing the Xilinx Research Labs Asia Pacific (XLAP). This work was mainly completed before his on-leave to XLAP. His research interests include network measurement, data center networking, and software defined networking. He is the recipient of a fellowship from the European Research Consortium for Informatics and Mathematics (ERCIM), New Century Excellent Talents in University awarded by the Ministry of Education, China.