Towards A Fast Packet Inspection over Compressed HTTP Traffic

Xiuwen Sun, Kaiyu Hou, Hao Li, Chengchen Hu
Department of Computer Science and Technology
Xi'an Jiaotong University
Email: {silvin, immike}@stu.xjtu.edu.cn, {hao.li, chengchenhu}@xjtu.edu.cn

***Abstract*—Matching multiple patterns is the key technology in firewall, Intrusion Detection Systems, *etc*. However, most of the web services nowadays tend to compress their traffic for less transferring data and better user experience, which has challenged the multi-pattern matching original working only on raw content. Naive and straightforward solutions towards this challenge either decompress the compressed data first and apply legacy multi-pattern matching methods, or have to scan redundant data during the matching., which are not fast and memory efficient. In this paper, we propose COmpression INspection (COIN) method for multi-pattern matching on compressed HTTP traffic. COIN does not decompress the data before matching and only scans once each bit of the traffic under inspection. We have collected real traffic data from Alexa.com top 500 and Alexa.cn top 20000 web sites and have performed the experiments under 1430 SNORT patterns. The evaluation results show that COIN is 10–31% faster than state-of-the-art approach.**

***Index Terms*—Deep Packet Inspection, Compressed Traffic, Multi-Pattern Matching, gzip/DEFLATE.**

I. Introduction

Intrusion Detection System (IDS), Intrusion Detection Protection (IDP), firewalls leverage multi-pattern matching technology to inspect the network traffic payload. The task becomes more challenging because of a more and more significant trend that the web service traffic today is transmitted after compression.

It was shown in [1] that 66% of Alexa top 1000 sites used HTTP compression in July 2010 and the percentage was increased to 95% for the top 500 sites in October 2016 [2]. This phenomenon makes the *lazy* method not acceptable, which simply ignores the compressed traffic – it is easy for an adversary to bypass the detection by compressing the anomaly traffic. A *naive decompression-and-match* approach used by most of the products match patterns after a decompression processing first. Obviously, this approach is as accurate as the matching on raw traffic without compression but it would be quite slow and hungry for memory. *ACCH* is a state-of-the-art method, which also activates matching after decompression but accelerates the entire process by reusing the saved information during the decompression phase for the matching phase [3]. We observe that ACCH has the redundant processing in the decompression phase and the matching phase, which is to be prevented in this paper.

After briefly reviewing the background and problem with ACCH in Section II, we propose a method called COmpression INspection (COIN) in Section III, which does not need to match again a same pattern in the compressed traffic if it has been appeared before. The basic idea to achieve this is based on the simple observation: the compressed encoding of a segment of bytes should always be the same associated with a same corresponding plain text before the compression, so there is no need to recheck the pattern within compressed data if it has been matched before. The experiments are described in Section IV and our evaluation results demonstrated that COIN further accelerates ACCH by 10-31% under real traffic data. Related works are presented in Section V and finally in Section VI, we conclude the paper.

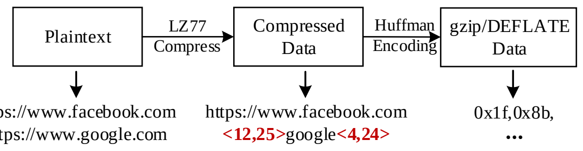

Fig. 1. Compressing process in gizp and DEFLATE

II. Problem

A. Background knowledge on gzip/DEFLATE

We focus on gzip in this paper [4], which is a compression encoding format recommended by HTTP 1.1 [5] and is utilized by more than 85% of the web sites as the default format shown by our experiment on Alexa Top sites. gzip uses DEFLATE [6] as its compression method built based on a combination of the LZ77 algorithm and Huffman coding.

Figure 1 shows an example of compressing with gzip. In the first stage, LZ77 algorithm compress the plain text *"https://www."* in the second line to be *"<12,25>"*, which means "go back 25 characters (line feed included) and copy 12 characters". The *"https://www."* in the first line is called ***referred string*** and the *"<12,25>"*, which is a two-tuple of *"<length, distance>"*, is named as ***pointer***.

The data encoded by Huffman coding, is a continuous bit stream with variable length. Therefore, the encoded data cannot be byte-coded, which is the reason why the previous work have to decompress traffic before a pattern matching method could be applied.

B. ACCH Algorithm

ACCH is fundamentally a **multi-pattern matching** approach based on Aho-Corasick (AC) algorithm [7]. An AC algorithm first constructs a Deterministic Finite Automation

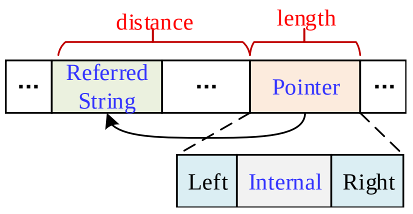

Fig. 2. Illustration of *pointer* and its *referred string*.

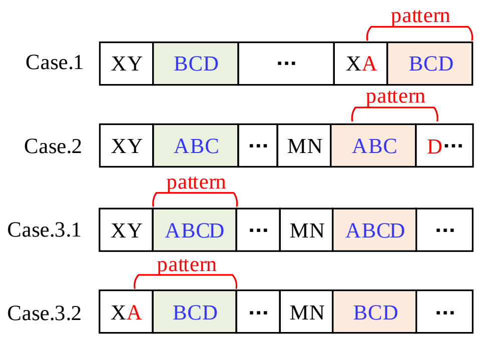

Fig. 3. Categories of ACCH.

(*DFA*) by patterns and then processes the input characters in a single pass with deterministic performance.

The basic observation in ACCH is: "If *referred string* does not completely contain matched patterns then the *pointer* contains none" [3]. Following this observation, *pointer* is separated to three parts (Figure 2). If no matched patterns occurred within the *referred string*, ACCH skips the *Internal* area in *pointer*. However, if not, ACCH has to scan those bytes again (it has scanned in its *referred string*).

ACCH divides the matching cases into three categories, as shown in Figure 3.

(1) *Left Boundary*: the pattern starts prior to the pointer and its suffix is in the pointer. For handling this case, it should determine whether a pattern ends within the pointer left boundary. ACCH uses *depth*, the number of edges on the shortest simple path from current scanning state to the DFA root, to indicate the longest prefix of any pattern within the pattern-set [8] at the current scan location. The algorithm continues scanning the pointer as long as *the number of bytes scanned* within the pointer is smaller than *depth*, otherwise moving to *Internal* case.

(2) *Right Boundary*: the pointer contains a pattern prefix and its remaining bytes occur after. By using a *status* parameter, a byte with u (*Uncheck*) means its *depth* is smaller than (not including) a pre-defined constant parameter *CDepth* (2 as default), otherwise the byte is marked c (*Check*). Then, ACCH locates the last occurrence of *Uncheck* within the *referred string* as *unchkPos* and scans from the corresponding position within the pointer. In order to determine safely the right boundary, the scan is resumed from *unchkPos - CDepth + 2*. In this case, it needs to pre-configure parameter *CDepth*. With different values of *CDepth*, the numbers of skipped bytes are influenced and may vary the performance.

(3) *Internal area*: a pattern ends within the referred string. ACCH introduces another status m (*Match*). In referred string, a byte with *Match* status means a pattern ends at its location. The same as Case 2, algorithm starts scanning at *unchkPos - CDepth + 2* position. In this case, *unchkPos* is the last occurrence of *Uncheck* prior the byte with *Match*. However, ACCH have to check again the complete pattern bytes if comes to again after its first scanning in its *referred string* before.

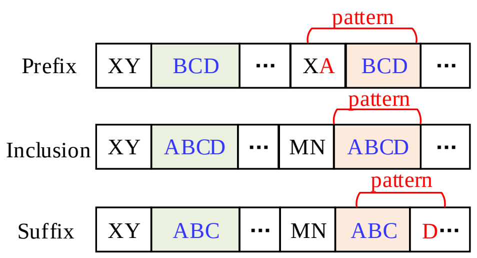

Fig. 4. Categories of COIN. The left shadow area represented *referred string* and the right represented *pointer*.

Example of ACCH with input text is presented in §III-B.

III. COIN

In this section, we propose a faster method COIN for multi-pattern matching on compressed HTTP traffic. COIN leverages different ways to match patterns according three cases as shown in Figure 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.

Please note that COIN uses a different case classification and different processing method as what ACCH does in *inclusion* and *suffix* cases. COIN can skip the bytes within *pointer* when scanning even if it contains a complete pattern.

A. Algorithm

Since the contents of *pointer* and *referred string* are identical, we have another hypothesis that if there is a complete pattern in *referred string*, then the pattern must also be in *pointer*.

COIN skips as many bytes as possible in *pointer* to accelerate multi-pattern matching. So, we need to handle the positional relationship between *pointer* and pattern correctly. As shown in Figure 4, COIN handles the three cases as the following.

(1) *Prefix*: COIN uses the same parameter *depth* defined in ACCH. It ends this procedure directly and moves to process case of *inclusion*, if *depth* equals to 0 at the prior byte of *pointer*. Otherwise, it would continue scan the *pointer* bytes as long as the number of scanned is smaller than *depth*.

(2) *Inclusion*: COIN introduces a parameter called *status*. A byte with *m (Match)* state means that a pattern is matched at that location. If patterns matched during the matching process, COIN records pattern's length and location as matching information. When handling *inclusion* case, COIN copies *status* from *referred string* to *pointer* first. Then let *mPos* be the position in the *pointer* with *Match* state, 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

Algorithm 1: COIN (COmpression INspection)

```
definition : $byteInfo$ - {byte, status, depth}
             $byteList$ - array of $byteInfo$
input      : $Trf_1...Trf_n$ - compressed traffic
function COIN($Trf_1...Trf_n$)
    for $i \leftarrow 1$ to $n$ do
        if $Trf_i$ is not pointer$(len, dist)$ then
            $byteInfo.byte = Trf_i$; AC_ScanByte$(byteInfo)$;
            $byteList$.Add$(byteInfo)$ ;
        else
            $byteList$.Add$(byteList[i - dist : i - dist + len])$;
            COIN_ScanPointer$(i, dist, length)$;

function COIN_ScanPointer$(i, dist, len)$
    $curPos \leftarrow 0$; $offset \leftarrow i - dist$;

    //=== 1. prefix case ===
    if $byteList[i-1].depth \neq 0$ then
        for $curPos \leftarrow 0$ to $len$ do
            if AC_ScanByte$(byteList[i + curPos]) \leq curPos$
              then break;
    if $curPos \geq len - 1$ then break;

    //=== 2. inclusion case ===
    for $j \leftarrow curPos$ to $len$ do
        if $byteList[i + j].status = MATCH$ then
            //pointer contains entire pattern
            if $match[i + j].len \leq j + 1$ then
                $curPos \leftarrow j + 1$;
                Record $match[i + j].len$ and $i + j$;
            else
                $byteList[i + j].status \leftarrow 0$;

    //=== 3. suffix case ===
    $lastByteInfo \leftarrow byteList[i + len - 1]$;
    if $curPos < len \wedge lastByteInfo.depth \neq 0$ then
        AC_ResetScanStatus();
        $offset \leftarrow i + len - lastByteInfo.depth$;
        while $offset \leq i \wedge offset < len$ do
            AC_ScanByte$(byteList[offset + curPos].byte)$;
```

Plaintext:	23445677789aa1234bb77890cc3345676dd3567
Compression:	23445677789aa1<3,14>bb<4,12>0cc3<5,26>6dd3<3,32>
Patterns:	123，890，4567

Fig. 5. Input data of example. There are 4 <length, distance> pairs in the compression data.

Ex.1	234	4567···aa1	234	bb···
depth	001	1234···001	231	00···
ACCH	uuu	uccm···uuu	cmu	uu···
COIN	000	000m···000	0m0	00···

(a)

Ex.2	···7	7789	aa···bb	7789	0···
depth	···4	0012	00···00	0012	3···
ACCH	···m	uuuc	uu···uu	u?uc	m···
COIN	···m	0000	00···00	0000	m···

(b)

Ex.3	2	34567	7 ··· cc3	34567	6···
depth	0	01234	0 ··· 000	01234	0···
ACCH	u	uuccm	u ··· uuu	uuccm	u···
COIN	0	0000m	0 ··· 000	0000m	0···

(c)

Ex.4	2344	567	7···dd3	567	···
depth	0001	234	0···000	000	···
ACCH	uuuu	ccm	u···uuu	uuu	···
COIN	0000	00m	0···000	000	···

(d)

Fig. 6. Example of COIN and ACCH

pattern is appeared or not. If so, COIN keeps the pattern's length and location, otherwise eliminates the *Match* state at *mPos*.

(3) *Suffix*: Copying *depth* to the pointer and determining whether a *suffix* pattern is presented by detecting *depth* of the last byte (*lastPos*) in the pointer. If *depth* is larger than 0, the DFA scans from the *lastPos-depth* position.

The detail of COIN is shown in Algorithm 1.

B. Example

We use the compressed data in Figure 5 as input to compare the pattern matching process in ACCH and COIN. In each sub-figure of 6, the first line is part of plaintext, the second line represents the *depth* of each character in DFA and the last two lines (ACCH and COIN) represent the *status* defined by their algorithms.

(a) *Prefix*: COIN shares the same process to scan *pointer* in Ex.1 with ACCH. COIN first copies *depth* and *status* from corresponding *referred string* to *pointer*. Because the *depth* of the byte ("1") prior to *pointer* is 1, it continues scanning bytes in *pointer* until ending at the last byte ("4") and matching a pattern ("123"). However, COIN doesn't skip any byte in this case, as the string in *pointer* has no more byte.

(b) *Suffix*: In Ex.2, COIN copies *depth* and *status* from corresponding *referred string* to *pointer*. The *depth* of the byte ("b") prior to *pointer* is 0, so the process moves to the *inclusion* case. However, there is not a *Match* state in *pointer*. Therefore, the process moves to the *suffix* case. The *depth* of the *lastPos* (3) is 2, so COIN begins a new scanning after the position of *lastPos-depth* ("8" at 2) to *lastPos*. In this process, COIN skips 2 bytes in pointer.

In ACCH, the *unchkPos* in *Right Boundary* case is 2 ("1"). It starts scanning form *unchkPos - CDepth + 2* (we use the best *CDepth*, 2, here). ACCH skips 2 bytes in this pointer. However, we usually cannot meet the best pre-defined *CDepth*.

(c) *Inclusion*.1: After process of *prefix* case, COIN finds a byte with *Match* state in *pointer* in which *mPos* is 4 (position

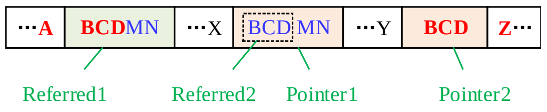

Fig. 7. Multi-referred string

of "4" in *pointer*) and the *mLen* is 4 (||"4567"||). Therefore, there is an entire pattern in this *pointer*. Then, the COIN starts scanning posterior byte with *Match* state. However, there is no more byte in this *pointer*. In this case, COIN skips 5 bytes.

In ACCH, the *unchkPos* is 1 ("4" is the nearest byte with *Uncheck* state prior to the byte "7" with *Match* state). It begins scanning from *unchkPos* and skips 1 byte.

(d) *Inclusion*.2: COIN notices a byte with *Match* state ("7") in *referred string*. However, for this byte, the *mPos* is 2 and *mLen* is 4. Therefore, there is not an entire pattern in *pointer*. COIN eliminates the *Match* state and ends scanning in *pointer*. COIN skips 3 bytes in this case.

ACCH cannot find a byte with *Uncheck* state and set *unchkPos* as 0. It scans the whole pointer and skips nothing.

C. Discussion

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 IV-B shows that over 90% of patterns are presented in *inclusion* case. Skipping scanning bytes in *inclusion* can bring lots of performance improvements.

In terms of memory usage, both of COIN and ACCH store the matching information.

COIN needs more space to preserve the parameter of *depth*, but the total memory requirements of COIN and ACCH are similar since ACCH has more information (Uncheck, Check or Match) to be stored than COIN.

Obviously, these three cases designed in COIN are basic and complete. They are only the positional relationships between *pointer* and pattern. And they can be assembled to all other more complex cases.

The numbers of bytes that can be skipped in ACCH are not the same under different *CDepth*. For example, when *CDepth=4* in Figure 6 (b), ACCH will begin scanning bytes with the second "7", which should be skipped under a well-chosen user-defined parameter. Moreover, if under an inappropriate parameter, it would scan some redundant bytes.

Figure 7 shows the example of multi-referred string. Pointer1 and Referred2 have some overlapped bytes. Since the scanning process will skip the first three bytes in Pointer1 (pattern "ABCD"), Pointer2 cannot get the *depth* value exactly from Referred2. Therefore, ACCH has to use *CDepth* to represent the *depth* approximately.

In *prefix* case of COIN, we only use the *depth* of one byte prior to pointer. And we don't use *depth* in *inclusion* case. Thus, we only need to discuss the use of *depth* in *suffix* case. Assume "BCDZ" is a pattern which needs to be matched in Pointer2, this pattern will be skipped and lost only if the *depth* of last byte in Pointer2 ("D") is less than 3. In other words, it occurs only if "BCD" has not been scanned in Pointer2 or Referred2. However, COIN will copy *depth* from its corresponding *referred string* (Referred1) to the unscanned *pointer* (Pointer1). Therefore, if "BCD" is a ***prefix*** of a pattern, Pointer2 will get exact *depth* (3) value from Referred1 at last. Otherwise, if "BCD" is not a ***prefix*** of a pattern, the *depth* will be larger than its actual value(4 in this example). 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 pattern in case of *suffix*.

The status values in COIN are also accurate in multi-referred situation. Assume "ABCD" is the pattern in Pointer1. The third byte ("D") has a *m* status. COIN first copies *status* value from Referred1 to Pointer1. After finishing processing *inclusion* case, the bytes' status in Pointer1 will be updated to "0" immediately. So, Pointer2 will gets accurate *status* from Referred2.

TABLE I
CHARACTERISTICS OF EXPERIMENTAL DATA SETS

	Alexa.com	Alexa.cn
Count of Pages	428	13747
Compressed Size (MB)	14.73	226.95
Decompressed Size (MB)	68.28	1190.99
Ratio of bytes represented by pointers	91.35%	91.92%
Average pointer length(B)	15.30	19.84

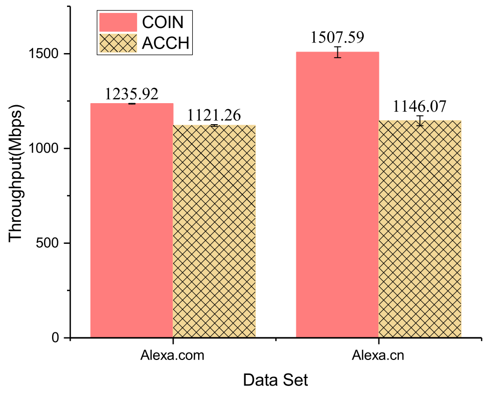

Fig. 8. Multi-pattern matching throughput of COIN and ACCH

IV. Evaluation

A. Settings

First, we have collected traffic by accessing the Alexa top sites as shown in Table I. Especially, all the raw traffic data related to the transferring of TOP 500 Alexa.com [2] (only top 500 is publicly available) and top 20000 Alexa.cn [9] sites using compression are used as the input traffic in our experiments. In addition, we take 1430 matching patterns from Snort rules [10] and use a desktop PC (Intel 4-core 3.4GHz and 8G RAM) for evaluation.

B. Performance

Figure 8 compares the performance of COIN and ACCH (*CDepth* = 2) for multi-pattern matching. COIN is more efficient than ACCH in both data sets. Particularly, COIN can process 1507Mb compressed traffic in one second and achieve 31% more throughput than ACCH on the Alexa.cn date set.

TABLE II
CHARACTERISTICS OF POINTER, PATTERN AND COMPARING PERFORMANCE

Data Sets	**Average pointer length(B)**	**Bytes represented by pointer**	**Pattern in inclusion ratio**	**Performance COIN/ ACCH**
Alexa.com	15.30	91.35%	92.61%	110.23%
Alexa.cn	19.84	91.92%	94.23%	131.54%
Subset	19.19	91.53%	93.64%	128.64%

As shown in Table II, most of matched patterns are entirely contained in *inclusion* case (**pattern in inclusion ratio** higher than 90%). **Average pointer length** in Alexa.cn data set and its subset (last 6000 websites in 13747) are longer than that in Alexa.com. Our evaluation shows that COIN achieves a higher performance in this data set than the data set of Alexa.com comparing to ACCH, which is mainly due to COIN can skip more bytes when scanning pointers and thus save much time on handling the inclusion case.

V. Related work

A. With gzip/LZ77

We have described ACCH in Section II and thus is omitted here. Besides, there are other work that focus on inspection the HTTP traffic compressed by LZ77. It was proposed in [1] that 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. [11] mainly focused on regular expression matching and retrieve to ACCH on string matching. The two papers studied in [12] and [13] to match on Huffman-encoded data, but they only applied to single-pattern matching instead of multi-pattern matching.

B. With other compression methods

It is achieved in [14] that multi-pattern matching on compressed data with LZW, but it cannot work on LZ77 compression algorithm and thus cannot support inspections over HTTP traffic. In [15], the authors applied the Boyer-Moore algorithm [16] to compress and traffic as well as the pattern for fast matching. However, the compressing method is (at least) not adopted by any Alexa top sites. In other words, it fails to inspect the web traffic nowadays. In addition, Google proposed a compression method called SDCH [17], 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 [18], which can make decompression-free inspection on the traffic compressed by SDCH is also limited since it cannot be extended to LZ77.

VI. Conclusion

Compression of HTTP traffic nowadays is popular, especially for the TOP web sites who generates the majority of the HTTP traffic. This phenomenon has introduced a requirement on the fast pattern matching within middleboxes. In this paper, we have presented a method called COIN for multi-pattern matching on compressed HTTP traffic. Not as what the previous work did to check the same pattern every time it appears in the compressed data segment, the proposed COIN only does the matching once and is able to skip the processing on its future appearance. The comparisons with state-of-the-art approach ACCH show a further 10-31% improvement in speed under the experiments with real traffic from Alexa .com top sites and .cn top sites. There is no need for any parameter settings for COIN, while ACCH's parameter configurations may vary the performance in different traffic scenarios.

VII. Acknowledgement

This work is supported by the National Key Research and Development Program of China (2016YFB0800101), the NSFC (No.61672425, 61402357), the Microsoft Research Asia Collaborative Research Program(No.2016JM6066), Fundamental Research Project of Natural Science in Shaanxi Province(No.2016JM6066) and the Fundamental Research Funds for the Central Universities.

References

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

[2] "Alexa top 500 global sites," "http://www.alexa.com/topsites/", accessed Oct. 2016.

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

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

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "rfc 2616, hypertext transfer protocol–http/1.1," https://www.rfc-editor.org/rfc/rfc2616.txt, June 1999.

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

[7] A. V. Aho, "Efficient string matching: an aid to bibliographic search," *Communications of the Acm*, vol. 18, no. 6, pp. 333–340, 1975.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 6.

[9] "Alexa top china sites," "http://www.alexa.cn/siterank/", accessed Feb. 2017.

[10] "Snort community rules," "https://www.snort.org/", accessed Dec. 2016.

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

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

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

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

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

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

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

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