This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE INFOCOM 2009 proceedings.

Accelerating Multi-Patterns Matching on Compressed HTTP Traffic

Anat Bremler-Barr
Computer Science Dept.
Interdisciplinary Center, Herzliya, Israel
Email: bremler@idc.ac.il

Yaron Koral
Computer Science Dept.
Interdisciplinary Center, Herzliya, Israel
Email: koral.yaron@idc.ac.il

***Abstract*—**

One of the fundamental technique which is used today by network security tools to detect malicious activities is 'signature based' detection. Today, the performance of the security tools is dominated by the speed of the string-matching algorithms that detect these signatures. Currently these security tools do not deal with compressed traffic, which becomes more and more common in HTTP. HTTP protocol uses the GZIP compression, which first requires some kind of decompression phase before performing the multi-patterns matching task. Thus, there is a high performance penalty in pattern matching on compressed data.

In this paper we present a novel algorithm, Aho-Corasick-based algorithm for Compressed HTTP (ACCH) that takes advantage of information gathered by the decompression phase in order to accelerate the commonly used Aho-Corasick pattern matching algorithm. We show by analyzing real HTTP traffic and real WAF signatures patterns, that we can skip scanning up to 75% of the data. Surprisingly, we show that in some situations, it is faster to do pattern matching on the compressed data, with the penalty of decompression, than doing pattern matching on regular traffic. As far as we know we are the first paper, that analyzes the problem of 'on-the-fly' multi-patterns matching algorithms on compressed HTTP traffic and suggest a solution.

I. Introduction

One of the fundamental techniques which is used today by security tools such as network intrusion detection system (NIDS) or web application firewall (WAF) to detect malicious activities is 'signature based detection'. In this technique, the security tools alert when signatures, a defined set of patterns of malicious activities, appear in the traffic. Today, the performance of the security tools is dominated by the speed of the string-matching algorithms that detect the signatures [1].

HTTP compression, also known as content encoding, is a publicly defined way to compress textual content transferred from web servers to browsers. Most popular sites and applications use HTTP compression, such as Yahoo!, Google, MSN, YouTube and Facebook. This standards-based method of delivering compressed content is built into HTTP 1.1, and most modern browsers that support HTTP 1.1 support GZIP compression [2]. On average, content encoding saves around 75% of text files (HTML, CSS, and JavaScript) and 37% overall [3]. Compressed HTTP is usually used on the response side from server to client due to the fact that generally responses contain most of the data while requests usually contain a short URL string.

Multi-patterns matching on compressed traffic is a difficult problem, since it requires two time-consuming phases: traffic decompression and pattern matching. Currently most security tools do not deal with compressed traffic. In some of the cases they just do not scan compressed traffic, which may be the cause of miss-detection of malicious activity. In other cases, the security tools ensure that there will not be compressed traffic by re-writing the HTTP header between the original client and server. In this technique, the security tools change the client side response in such a way that it indicates that compression is not supported by the client's browser. This method harms the performance and bandwidth of both client and server for the specific connection. The few security tools that handle HTTP compressed traffic usually work in a proxy mode, meaning that they construct the full page on the proxy by decompressing it, after that perform a signature scan and then forward it to the client. The last option is not applicable in security tools that cope with high speeds and in case where inserting a delay is not a real option.

In this paper we present a novel algorithm, Aho-Corasick-based algorithm on Compressed HTTP (ACCH). The algorithm is based on the following surprising observation: we can take advantage of the fact that we deal with compressed traffic to accelerate the multi-patterns matching phase. Specifically, we use the fact that the GZIP compression algorithm, works by eliminating repetitions of strings using back-references (pointers) to the repeated strings. Our key idea is to store information produced by the pattern matching algorithm, for the already scanned uncompressed traffic, and then in case of pointers, to use this data in order to understand if there is a possibility of finding a match or we can skip scanning this area. We show that by using this information, we can skip up to 75% of the data and gain up to 70% improvement in the performance of multi-patterns matching algorithm. As far as we know we are the first paper that analyzes the problem of 'on-the-fly' multi-patterns matching algorithm on compressed HTTP traffic, and suggest a solution.

II. Background

The GZIP algorithm: One of the HTTP 1.1 recommended compression algorithms and the most commonly used is the GZIP algorithm [4] [2]. GZIP is based on the DEFLATE algorithm that compresses the file using combination of the

978-1-4244-3513-5/09/$25.00 ©2009 IEEE

following two compression techniques: the text is compressed by the LZ77 algorithm. Then, the output is compressed by the Huffman coding. We elaborate on the two stages:

LZ77 Compression [5]- The basic idea of the LZ77 compression technique is that we can compress a series of bytes (characters) if we spot that this series of bytes has already appeared in the past (specifically, in the sliding window of the last 32KB of uncompressed data). In such a case we encode this series of bytes (denoted by repeated string) by the pair ($distance$,$length$) where $distance$ is a number between 1-32768 (32KB) indicates the distance in bytes of the repeated string and $length$ is a number between 3-258 indicates the length of the string in bytes. For example, the text: "abcdefabcd", will be compressed to: "abcdef(6,4)", i.e., return 6 bytes and copy 4 bytes from that point.

Huffman Coding [6]- In HTTP, Huffman encodes uncompressed bytes and pointers (i.e., as numbers). The Huffman dictionary is usually added to the beginning of the compressed file (otherwise a predefined dictionary is selected).

Multi-patterns matching: Pattern-matching has been a topic of intensive research that has resulted in several approaches. The two fundamental approaches are the Aho-Corasick [7] and the Boyer-Moore [8]. Aho-Corasick (AC) algorithm constructs a finite state machine (FSM) for detecting all occurrences of a given set of patterns by processing the input in a single pass, performing a state transition for each input byte. In this paper we illustrate our technique using the Aho-Corasick algorithm, however as a part of our future work, we investigate the usage of our suggested technique with some other pattern matching algorithms.

III. The challenges in performing multi-patterns matching on compressed HTTP traffic

In this section we give an overview of the obstacles in performing multi-patterns matching on compressed HTTP traffic. Note, that dealing with web live traffic is much more challenging than the task of offline multi-patterns matching on compressed data. First, the compression method cannot be chosen or modified. Second, pattern matching should be performed '**on-the-fly**' on the ongoing web traffic.

We note that there is no "easy" way to perform multi-patterns matching over compressed traffic without decompressing the data in some way. The main reason for this is that $LZ77$ is an $adaptive$ compression algorithm due to the use of back-references pointers. In an adaptive compression, the text represented by each compression symbol is determined dynamically by the data. As a result, the same substring will be encoded differently depending on its location in the text. Thus, decoding the pattern is futile since it will not appear in the compressed text in some specific form. For example, consider the case where we search for pattern "abcd". The pattern can be expressed in the compressed data by $abc*^{j}\ (j+3,3)d$ for all possible $j < 32765$. On the other hand, Huffman encoding, is non-adaptive within a given text and the same pattern will always be encoded to the same bit string. However, since the LZ77 part is adaptive, the combination of the two algorithms is therefore adaptive.

The naive (direct) way of performing multi-patterns matching on the traffic, in real time, as required by security tools is by combining the following steps (See Algorithm 1):

1) Remove the HTTP header and store the Huffman dictionary of the specific session in memory. Note that different HTTP sessions would have different Huffman dictionaries.
2) Decode the Huffman mapping of each symbol to the original byte or pointer representation using the specific Huffman dictionary table.
3) Decode the LZ77 part.
4) Perform multi-patterns matching on the uncompressed traffic.

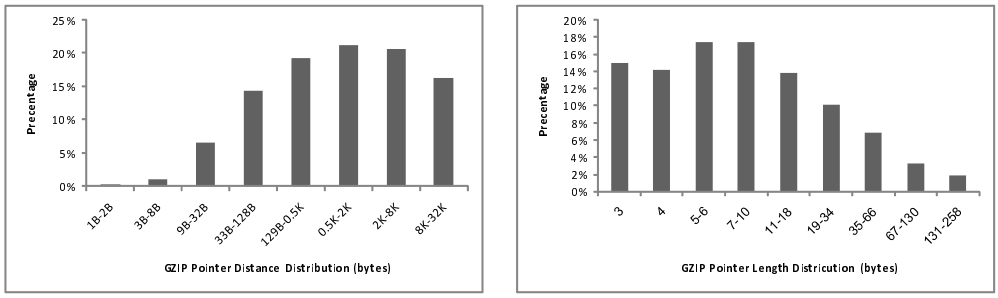
The challenges in the multi-patterns matching algorithm on compressed traffic are both from the space and time aspects:
Space - One of the problems of decompression is its memory requirement; the straight forward approach requires a 32KB sliding window for each HTTP session. Note that this requirement of storing 32KB of the uncompressed data is difficult to avoid, since the back-reference pointer can refer to any point in the 32KB sliding window and the pointers may be recursive unlimitedly (i.e., pointer may point to area with a pointer). Figure 1(a) shows that indeed the distribution of pointers on real life data set (see Section VI for details on the data set) is all around the 32KB sliding window. On the other hand, pattern matching of non-compressed traffic requires only storing one or two packets (to handle cross packet data), where an average TCP packet is around 1.5KB. Hence, the fact that we are dealing with compressed traffic poses a higher memory requirement by a factor of 10. Therefore in order to handle compressed traffic, a Mid-Range firewall that handles 30K concurrent sessions, needs 1GB memory while a High-Range firewall that handles 300K concurrent sessions, needs 10GB memory. This memory requirement, has implication on not only on the price and feasibility of the architecture but also on the capability to perform caching.

Time - Recall that AC scan-time is a dominant factor in the performances of security tools [1]. We introduce here a simple model that will help us to compare the time requirement of the decompression with the time requirement of the AC algorithm.

A key influence on the time is the ability to perform fast memory references to the limited memory of the cache as opposed to the slower main memory. We assume in our model, that we can choose that some of the data structures will be in cache memory. [1]

Let M be the cost of one memory reference, C the cost of one cache reference, P_l the average length of a pointer in the compressed traffic and P_r the fraction of bytes represented by pointers in the compressed traffic. Let B be the cache block size which is typically 32 Bytes in SDRAM memory (i.e.,

[1]Using hardware solutions or by special assembly commands that give recommendation to the loader.



(a) (b)

Fig. 1. Distribution of the following pointer characteristics on the real life data set of Section VI (a) distance of a pointer (b) length of a pointer.

each memory lookup brings to the cache 32 Bytes from the surrounding area of the memory address). We show that B has a dramatic influence on the performance of the decompression algorithm since the pointers refer to consecutive addresses in the memory.

We start by analyzing AC algorithm. The FSM can be represented by a two dimensional matrix (referring the Snort implementation [9]), where a row represents a state and a column represents an input byte. Each state has also a match pointer that points to a matched pattern list if this is a "match state" (a.k.a. "output state" [7]) or NULL otherwise. For every byte, the AC algorithm performs two memory references, one for next state extraction and the other to check whether a match was found. [2] If the FSM is not in cache memory due to its large size (for example 53.1MB for 1533 Snort patterns in 2003 [10]), the memory cost for each byte is $2M$. Otherwise, if the FSM fits into the cache, due to a small number of patterns or the usage of FSM compression techniques [11], [12], [13], [10], [14] the memory cost is around $2C$.

We now analyze the GZIP algorithm. GZIP maintains two data structures per HTTP session for the decompression phase: a small one for the Huffman dictionaries (less than 700 bytes, which is small enough and therefore assumed to be in cache) and a larger one for the 32KB sliding window. Note that naive Huffman decoding is made by one bit at a time. [3] Worst case analysis approximates the number of memory lookups required by Huffman decoding as the number of bits in the compressed file divided by the actual bytes in the uncompressed file. Considering an average compression ratio of 75%, this indicates that there are 2 memory lookups per byte in the uncompressed file. Since the Huffman dictionary is small enough to fit into the cache, we approximate the Huffman overhead by $2C$.

We now analyze the LZ77 decompression part. For a small number of concurrent sessions where all of the 32KB sliding windows fit into cache, the cost per byte which is a not a pointer is $1C$ reference for updating the 32KB sliding window.

[2]If there is more than one matching pattern for a state then more memory accesses are required, but for simplicity this case is ignored. Note that each state (a row) is much bigger than the general block size, since usually each state holds data about transitions for each of the 256 possible inputs.

[3]We believe that Huffman decoding algorithm can be improved, and it is part of our future work.

	Cache	Memory
AC	$2C$	$2M$
GZIP decompression	$\sim 4C$	$\sim 5C$

TABLE I
SUMMARY OF THE ANALYSIS OF TIME COMPLEXITY OF AC AND GZIP DECOMPRESSION WHERE C IS CACHE LOOKUP TIME AND M IS MAIN MEMORY LOOKUP TIME

A byte represented by a pointer requires $2C$ for both reading and writing to the 32KB sliding window. Adding the additional 2 Huffman decoding references we are bounded by a total time of $4C$. However, if the number of concurrent sessions is high, then in the case of a pointer, we first need to bring the blocks to the cache. The average number of cached blocks retrieved per pointer, denoted by B_P, is $B_P = \lceil (P_l/B) \rceil + (P_l mod B)/B$. In order to understand the cost per byte we need to divide the result by the average pointer length. Hence we need total $P_r * (B_P/P_l * M + 4C) + (1 - P_r) * 2C$ time. Using a real-life data set, we retrieved the value of average P_l as 17.8 (Figure 1(b)); see Section VI for details on the data set. By examining the same data set, we retrieve also the ratio of bytes represented by pointers, P_r as 0.921. Since today the factor between the regular memory lookup to cache memory lookup of M/C is usually between $10 - 20$ and B is $32Bytes$, the $P_r * B_P/P_l$ can be bounded by $0.08M$ which is around $1C$. Hence the fact that number of concurrent sessions is high has a minor impact on the decompression time, because each pointer refers to consecutive bytes. Since most of the compressed file is represented by pointers (i.e P_r = 0.921), memory access time is dominated by the pointer analysis part, hence it is around $5C$.

Table I summarizes the findings. One outcome of this analysis is the observation that when the FSM is in the cache, the GZIP decompression has a higher time requirement than the AC algorithm itself. In the case where the FSM is in regular memory, the GZIP decompression takes only 20% of the time of the AC (assuming $M/C \sim 10$). In this paper we focus on AC performance. We show that we can reduce the AC time by skipping more than 70% of the FSM scans of the bytes and hence reduce the total time constraint for handling pattern matching in compressed traffic.

IV. Related Work

The problem of pattern matching on compressed data has received attention in the context of the Lempel-Ziv compression family [15], [16], [17], [18]. However, the LZW/LZ78 are more attractive and simple for pattern matching than the LZ77 compression algorithm. HTTP uses LZ77 compression, which has simpler decompression algorithm, but performing pattern matching on it is a more complex task that requires some kind of decompression (see Section II). Hence all the above works are not applicable to our case. The paper [19] suggested modification to the LZ77 compression algorithm in order to make the task of the matching easier in files. However, the suggestion is not currently implemented in today HTTP.

The paper [20] is the only paper we are aware of that deals with pattern matching over LZ77. However, in this paper the algorithm is capable of matching only one pattern and it requires two passes over the compressed text (file), which is not applicable for the problem in the network domains that requires 'on-the-fly' processing. Hence, as far as we are aware of, we are the first paper that deals with pattern matching on compressed HTTP traffic, i.e., on the LZ77 family, and in the context of networking.

One outcome of this paper, is the surprising conclusion that in some of the cases, pattern matching on compressed HTTP traffic with the overhead of decompression, is faster than performing pattern matching on regular traffic. We note, that in other compression algorithms (not LZ77 which is our case), a similar conclusion was shown in another context. The papers [21], [22], [23] show that compressing a file once and then performing pattern matching on the compressed file, accelerate the scanning process.

V. Aho-Corasick based algorithm for Compressed HTTP (ACCH)

In this section, we present our Aho-Corasick based algorithm for Compressed HTTP (ACCH). We start by giving a general description and intuition of the algorithm. As recalled, HTTP uses the GZIP compression, which its LZ77 part compresses data by using pointers to past occurrences of byte (character) sequences. Thus, the bytes that the pointer refers to in the sliding window (denoted by us as referred bytes) were already scanned for pattern matching and we can use this knowledge to save unnecessary scans.

Note that even if no pattern was matched during the scan of the referred bytes, we still need to rescan some bytes of the pointer. This is due to the fact that a pattern may occur at the boundary of the pointer. A prefix of the referred bytes may be a suffix of a pattern that started previously to the pointer and a suffix of the referred bytes may be prefix of a pattern that continues after (see Example 1, in Fig. 2). Moreover, in the case of a match at the referred bytes, we still need to check if the pattern occurs, since it might be the case where only the pattern suffix is referred by the pointer (see Example 2, in Fig. 2).

Intuitively, detecting a pattern at the left boundary can be done by continuing the scan up to a certain point of the pointer

Algorithm 1 Naive Decompression with Aho-Corasick pattern matching

Trf - the input, compressed traffic (after Huffman decompression)
$SWin_{1\cdots 32KB}$ - the sliding window of LZ77, where $SWin_j$ is the information about the uncompressed byte which is located j bytes before current byte
$FSM(state, byte)$ - AC FSM receives state and byte and returns the next state, where startStateFSM is the initial FSM state
$Match(state)$ - if state is "match state" it stores information about the matched pattern, otherwise NULL

```
1: state = function scanAC(state)
2: state=FSM(state,byte)
3: if Match(state) ≠ NULL then
4:     act according to Match(state)
5: end if
6: return state

7: procedure GZIPDecompressPlusAC(Trf_1 ··· Trf_n)
8: state=startStateFSM
9: for i=1 to n do
10:    if Trf_i is pointer (dist,len) then
11:        for j=0 to length-1 do
12:            state = scanAC(state, SWin_{dist-j})
13:        end for
14:        update SWin with bytes SWin_{dist···dist-len}
15:    else
16:        state=scanAC(state, Trf_i)
17:        update SWin with the byte Trf_i
18:    end if
19: end for
```

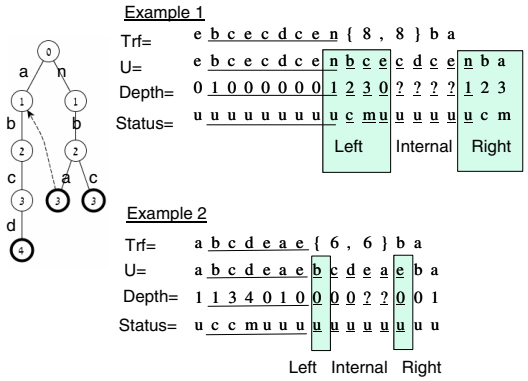


Fig. 2. Example of ACCH run for CDepth=2. The number inside every state indicates its depth. Question marks indicate that the corresponding bytes were skipped therefore their depth is not known to the algorithm. The pointer area is underlined by a dashed line. The referred bytes are underlined by a solid line.

(discussed later). Detecting a pattern at the right boundary or detecting a whole referenced matched pattern, requires storing information about the previous scans. One natural option is to store the state of the scanned byte at the FSM. However, this is not feasible since our main aim is to skip scanning some of the bytes, hence we cannot determine the exact state of those skipped bytes (which might be required later on in case of a pointer to a pointer). Luckily, we show that it is enough to store only the information of whether the state of a scanned byte is a Match and the information about the depth of the

scanned byte state at the FSM. The depth of a state s is defined as the number of edges in the shortest simple route between the start state to state s in the FSM. In order to understand how we can use the depth information, let us look at the case of a match at a referred byte. If we would have the exact depth information, we could check if the pattern occurs at the pointer area by comparing the length of the matched pattern, which is equal to the matched state depth, to the matched byte location at the pointer area. Once again, storing the exact depth information is not feasible, since we cannot calculate the depth of skipped bytes. Fortunately, we show that it is enough to store information which is a relaxation of the depth. We do that by storing information that estimates if the depth is below some threshold, denoted by $CDepth$, a constant parameter of our algorithm.

Specifically, we store a status for each byte in the sliding window. The status is an information about the state we reach at the FSM after scanning that byte. The status of a byte is coded by 2 bits and can be one of the following types: $Match$, $Check$ and $Uncheck$. $Match$ indicates reaching a "match state" at the FSM. The two other statuses of a byte indicate estimation on the depth of the scanned byte state in FSM in the case where the status is not $Match$. As recalled, the estimation is required, since we cannot calculate the state (and hence the depth) of the skipped bytes. Note, that we also aim to space-efficient status information. Our estimation is in two scenes. First, we don't store the exact depth. We define $CDepth$ as a constant parameter of the algorithm that represents "low depth". Its optimal value is determined using the experiments described on Section VI. Our status would indicate only if the depth is above or below $CDepth$. Second, we allow mistake in the estimation. The estimation can be wrong only in one direction where the estimated depth is deeper than it should be. Specifically, we define the status of a scanned byte as $Uncheck$ if the depth of the state is below $CDepth$. We mark the status as $Check$ if the depth *may* be $CDepth$ or above. The *may* is due to the fact that we might mistake estimating the depth of a byte by giving it a higher value than its actual depth in the cases of skipped bytes. As noted in [11], [12], most of the time the FSM uses states of low depth, hence in most cases the status of the bytes would be $Uncheck$, therefore we could skip most of the bytes.

We now give the details of the algorithm (see detailed pseudo code in Algorithm 2). We define $scanAC$ as a function that receives as an input the 'current state' (of the FSM) and a byte and return the 'next state' and 'byte status' by performing AC FSM transitions (see Lines 1-11).

In general, we need to handle three cases where there are possible occurrences of patterns: left boundary of pointer area, right boundary of pointer area and internal area, where patterns (can be multiple occurrences or none) are fully contained within a pointer area (see Example in Fig. 2).

Algorithm 2 Compressed HTTP based Aho-Corasick

Parameters as in the Naive Algorithm, with addition:
$SWin_{1\cdots 32KB}$ - Here $SWin_j$ the information about the j^{th} byte is a record of two: $SWin_j.b$ - byte $SWin_j.st$ -status
$Depth(state)$ - return the depth of the state in the FSM
$CDepth$ - the constant parameter of ACCH algorithm

```
1: (status,state) = function scanAC(state,byte)
2: state=FSM(state,byte)
3: if Match(state) ≠ NULL then
4:     act according to Match(state)
5:     status = Match
6: else
7:     if Dept(state) ≥ CDepth then status = Check
8:     elsestatus = Uncheck
9:     end if
10: end if
11: return (status,state)

12: procedure ACCH(Trf_1 ··· Trf_n)
13: state=startStateFSM
14: for i=1 to n do
15:     if Trf_i is pointer (dist,len) then
16:         j=0;
17:         while (Depth(state) > j)and(j < len) do
                                          ▷ Left Boundary scan
18:             (status,state)=scanAC(state,SWin_{dist−j}.b)
19:             update SWin with SWin_{dist−j}.b and status
20:             j++
21:         end while
22:         k=j-1
23:         while k < len − 1 do
                          ▷ Check Matches inside the pointer area
24:             Find the minimal k , j ≤ k < len
                    such SWin_{dist−k}.st = Match
25:             If no such k exist then k = len − 1
                                    ▷ Case of Right Boundary
26:             Find the maximal p, j ≤ p ≤ k
                    such SWin_{dist−p}.st = Uncheck
27:             If no such p exist then p = j
28:             if j < (p − CDepth + 1) then
                                    ▷ Skip bytes and update window
29:                 update SWin with SWin_{(dist−j)...(dist−p−CDepth+1)}
30:                 state=startStateFSM
31:                 for j=(p − CDepth + 1) to (p − 1) do
                                ▷ Scan bytes and copy status from SWin
32:                     (state,status)=scanAC(state,SWin_{dist−j}.b)
33:                     update SWin with SWin_{dist−j} and status
34:                 end for
35:             end if
36:             for l=j to k do
                                    ▷ Internal or Right Boundary scan
37:                 (state,status)=scanAC(state,SWin_{dist−l}.b)
38:                 update SWin with SWin_{dist−l}.b and status
39:             end for
40:             j=k+1
41:         end while
42:     else
                                                ▷ Byte Scan (no pointer)
43:         (state,status)=scanAC(state, SWin_{dist−i}.b)
44:         update SWin with SWin_{dist−i}.b and status
45:     end if
46: end for
```

Left Boundary: In order to detect a pattern that is in the left boundary of pointer area (see Lines 17-21), it is enough to continue scanning with $scanAC$, until we reach the j^{th} byte in the pointer area, where the depth of the state we reach in the FSM after scanning the j^{th} byte, is less or equal to j (the number of bytes in the pointer area we already scanned). There is no need to continue the scan, since from this point, if a pointer contains a pattern it would be fully contained within pointer area and this is dealt with in the next case.

Internal area: In order to detect patterns that are fully contained within a pointer area (Lines 23-41), we need to check if there is a byte with $Match$ status in the referred bytes. Let k be the first index where the k^{th} byte has a $Match$ status. Note that a $Match$ in the referred bytes indicates that there is a possibility of a pattern within the pointer area. Specifically, if there is a pattern within the pointer area, then there must be a $Match$ within the referred bytes, since we are now matching a pattern that is fully contained in the pointer area and hence, has been fully contained in the referred bytes. However, we still need to check, since we might have a $Match$ in the referred bytes for a pattern which only its suffix was pointed by pointer area.

We can now use the $Check/Uncheck$ status to determine how many bytes before k we need to scan. Let p be the the maximal index, such as the p^{th} byte has $Uncheck$ status and $p < k$. It is easy to see that if there is a pattern within this area, it can start only after the $p-CDepth+1$ position in the referred bytes, since otherwise we would have a contradiction to the definition of p. Hence we skip scanning bytes from j up to the position of $p - CDepth + 1$. This is the saving by our algorithm and the way we achieve the performance improvement. The challenging part is to maintain a correct status for the bytes we skip scanning (for use in case of future pointers to this area) without calling $scanAC$ on these bytes. The key idea, is to maintain statuses of these bytes from the corresponding bytes in the sliding window (see line 29). If a status of a byte in the pointer area is $Check$ then the corresponding byte in the referred bytes will be $Check$. This is due to the fact that in this part, the state that the FSM would have been, if we run the $scanAC$ on all the bytes, is correlated to a pattern that is internal to the pointer area and hence was also internal to the referred bytes. Note that the opposite is not true (i.e., a byte that has a status of $Check$ in the referred bytes may have a status $Uncheck$ in the corresponding byte at the pointer area, if we had run $scanAC$). A status of a byte in the referred bytes, may be due to a pattern that started before the referred bytes. Hence, in the case that we have a pointer to this pointer area, we might call $scanAC$ to correct status of bytes that were marked as $Check$ but where their true status should have been $Uncheck$, however these redundant calls do not harm the correctness of the algorithm.

In order to continue scanning, we set FSM to start state and continue scanning from $p - CDepth + 1$. Note that statuses for the area $p - CDepth + 1$ to $p - 1$ where maintained from $SWin$ (Lines 31-34). As a rule $scanAC$ gives more accurate information about the real status of the bytes, however, at these $CDepth$ bytes before p the status of $scanAC$ may be misleading - since we start from the start state of FSM, it will always return $Uncheck$ status even though it may not be the case. After the first $CDepth$ bytes we continue scanning up to the k^{th} position, this time we update statuses that returned from $scanAC$ (lines 36-39).

We then repeat the scan for patterns that are fully contained in the remaining pointer area, until there are none (line 23). In that case we check if there is a pattern at the right boundary of the pointer in a similar way to previous the case, we find the maximal index p such as the p^{th} byte has $Uncheck$ status and continue calling $scanAC$ from the $p - CDepth + 1$ position.

In the next theorem we prove the validity (correctness) of the algorithm.

Let P be a finite set of patterns, and Trf the compressed traffic.

Theorem 1: $ACCH$ detects all patterns in P in the decompressed traffic form of Trf.

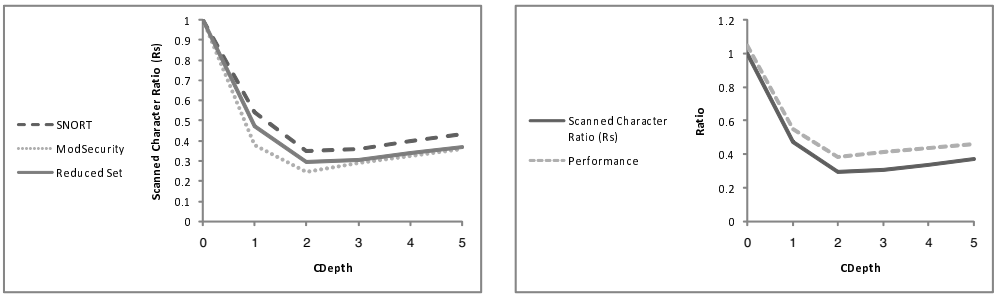
Sketch of Proof: The full detailed proof is given in the appendix. The proof relies on the validity of AC algorithm. We perform pattern matching on the compressed traffic twice, once with the naive algorithm (decompression + AC that scans all bytes), denoted as the $Naive$ algorithm, and another time with $ACCH$. The two algorithms use the same FSM, the only difference is that $ACCH$ skips scanning some of the bytes. In Lemma 4 in the appendix, which is the heart of the proof, we compare for every byte in the decompressed traffic the state and status, that each of the algorithms reached. We show that the three invariants claim holds: 1) If a byte has the status of $Check$ in the $Naive$ algorithm then it will also have the status $Check$ in the $ACCH$ algorithm. Note that the opposite direction does not hold. 2) Iff a byte has a $Match$ status in the $Naive$ algorithm, it will also have $Match$ status in the $ACCH$ algorithm. Note, that this is iff, i.e., the two directions hold. From this claim we can conclude that $ACCH$ detects all the patterns that $Naive$ detects and theorem follows. 3) Both algorithms, $ACCH$ and $Naive$ reach exactly the same FSM state after scanning of any single byte or after scanning a decompressed pointer entirely. This is not true within a pointer, since the $ACCH$ may skip scanning some of the bytes. The proof relies heavily on the characteristics of AC FSM. ∎

VI. Experimental Result

In this section, we evaluate the performance benefit of ACCH algorithm, and find the optimal $CDepth$, a key parameter of our algorithm, using real life traffic.

A. Data Set

In order to evaluate ACCH performance we need two data sets, one of the traffic and the other of the patterns. We use traffic that was captured by a corporate firewall for 15 minutes. The traffic contains 23,698 compressed HTTP-responses that take 910MB in uncompressed form. P_r, the ratio of bytes represented by pointers in these compressed files is equal to 0.921, and P_l, the average length of a the back-reference pointer is equal to 17.8.



(a) (b)

Fig. 3. (a) Scanned charcater ratio as function of $CDepth$ for ModSecurity, Snort and the Reduced Snort data-set (b) Performance Benefit as oppose to the Scanned charcater ratio (R_s) as function of $CDepth$ for the Reduced Snort data-set compared to the naive algorithm performance

As a data set of signatures, we use two sets: one of the ModSecurity [24], an open source web application firewall, and one of Snort, an open source network intrusion prevention and detection system [9].

In ModSecurity we choose a signatures group that apply to HTTP-response (only the response is compressed). Patterns are normalized in such a way that they will not contain regular expressions. [4] Total number of patterns is 124.

The Snort data-set, taken from the published rules on June 08 [9], contains more than 8K patterns. Most of them are in a binary mode and fit less to HTML searches. The binary patterns have no effect on the textual HTML files therefore, we removed them and were left with a subset of 1,202 textual patterns. We note that the patterns of Snort, are not designed to be applied on an HTTP-response, and hence are less applicable in the domain problem of compressed traffic. However, we decided to use this data set, since most pattern-matching papers refer to it, and since it gives us a chance to evaluate the effect of the number of patterns and number of matches on our algorithm performance. We note that Snort patterns occurs significantly more in traffic data, since Snort has patterns like "st", "id=" or "ver". The data set contains 476 occurrences of ModSecurity patterns and over $16M$ of Snort patterns.

B. ACCH performance

In this subsection, we compare the performance benefit using the ACCH algorithm. We define R_s as the scanned character ratio. R_s is a dominant factor for performance improvement for ACCH over naive algorithm as shown later. An important factor for ACCH is the $CDepth$ parameter. Figure 3 (a) summarizes R_s as a function of $CDepth$, for Snort and ModSecurity patterns. $CDepth$ equals 0 represents the $Naive$ algorithm. As shown in Figure 3, $CDepth$ 2 shows best performance for both pattern sets. R_s for Snort patterns is **0.35** and for ModSecurity is **0.24**. Note that increasing the value of $CDepth$ has two effects, on one hand the $CDepth$ may reduce the number of bytes that are marked as $Check$, but on the other hand it increases the number of bytes we need to scan before a byte whose status is $Match$ or $Check$. Snort patterns causes a significant amount of matches (near 2% of the total number of bytes scannes). This causes for much more scan areas and therefore much higher R_s. In order to check the influence of the number of matches on R_s we synthesized a reduced Snort data-set by removing the most frequent 88 patterns. The data contains $249,886$ matches (instead of 16M). As can be seen from 3 (a), removing only 88 patterns had a significant effect on the R_s value (R_s = 0.29 for $CDepth = 2$). Thus, match ratio has a much more significant influence than number of patterns on R_s value.

Figure 3 (b) shows the correlation between the R_s and the performance benefit. We use Dual Core Intel 1.8GHZ with 1Giga RAM as a platform. There is a slight overhead in implementing the algorithm, which is between 6%-10% that can be explained by the additional memory references to states of bytes causing a larger GZIP data structure. For $CDepth = 2$, ACCH running over ModSecurity achieved 69% performance improvement and ACCH running over Sbort achieved 61.4% performance improvement.

It is easy to see from the algorithm that two factors influence the scanned character ratio, the ratio of matches, which is usually low for security tools, and the ratio of "$Uncheck$", the number of bytes in the traffic that reach high depth in the FSM. As noted before and analyzed by [11], [12] most of the time the FSM uses states of low depth.

Combining the fact that ACCH has $60\% - 70\%$ performance improvement with the analysis of the cost of GZIP and AC (see Table I), the surprising finding is that we work faster on the compressed files than on uncompressed ones, especially when AC is in memory. ACCH improves overall performance also when FSM is in cache for about 20%.

VII. Conclusion and Future Work

At the heart of almost every modern security tool is a pattern matching algorithm. HTTP compression becomes very common in today web traffic. In some of the cases decompression causes a significant performance overhead to the overall pattern matching process. Our algorithm, achieves elimination of up to 75% of data scans based on information stored in the compressed data. Surprisingly, in some situations, it is faster to do pattern matching on compressed data, with the penalty of decompression, than doing pattern matching on regular traffic. Note that ACCH is not intrusive for the AC algorithm,

[4] Regular expressions were opened into several plain patterns.

therefore all the methods that improve AC FSM [11], [12], [13], [10], [14] are orthogonal to ACCH and are applicable. As far as we know we are the first paper, that analyzes the problem of 'on-the-fly' multi-patterns matching algorithms on compressed HTTP traffic, and suggest a solution. One open interesting question is how to refine the algorithm to maximize the gain from the ACCH approach. [5]

VIII. Acknowledgment

We would like to thank David Movshovitz, former VP security Technologies at F5 Networks and Ivan Ristic, VP of Security Research of Breach Security and the creator of ModSecurity, for helpful suggestions.

References

[1] M. Fisk and G. Varghese, "An analysis of fast string matching applied to content-based forwarding and intrusion detection," *Techical Report CS2001-0670 (updated version)*, 2002.
[2] P. Deutsch, "Gzip file format specification," May 1996. RFC 1952, http://www.ietf.org/rfc/rfc1952.txt.
[3] "Website optimization, llc." http://www.websiteoptimization.com.
[4] P. Deutsch, "Deflate compressed data format specification," May 1996. RFC 1951, http://www.ietf.org/rfc/rfc1951.txt.
[5] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, pp. 337– 343, May 1977.
[6] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of IRE*, p. 10981101, 1952.
[7] A. Aho and M. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, pp. 333–340, 1975.
[8] R. Boyer and J. Moore, "A fast string searching algorithm," *Communications of the ACM*, pp. 762 – 772, October 1977.
[9] "Snort." http://www.snort.org (accessed on July 2008).
[10] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memoryefficient string matching algorithms for intrusion detection," in *INFOCOM 2004*, 2004.
[11] T. Song, W. Zhang, D. Wang, and Y. Xue, "A memory efficient multiple pattern matching architecture for network security," in *INFOCOM 2008*, pp. 166 – 170, April 2008.
[12] J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications*, pp. 1–13, April 2006.
[13] V. Dimopoulos, I. Papaefstathiou, and D. Pnevmatikatos, "A memory-efficient reconfigurable aho-corasick fsm implementation for intrusion detection systems," in *Embedded Computer Systems: Architectures, Modeling and Simulation. IC-SAMOS*, pp. 186–193, July 2007.
[14] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network ids/ips," in *ICNP*, pp. 187–196, 2006.
[15] A. Amir, G. Benson, and M. Farach, "Let sleeping files lie: Pattern matching in z-compressed files," *Journal of Computer and System Sciences*, pp. 299–307, 1996.
[16] T. Kida, M. Takeda, A. Shinohara, and S. Arikawa, "Shift-and approach to pattern matching in lzw compressed text," in *10th Annual Symposium on Combinatorial Pattern Matching (CPM 99)*, 1999.
[17] G. Navarro and M. Raffinot, "A general practical approach to pattern matching over ziv-lempel compressed text," in *10th Annual Symposium on Combinatorial Pattern Matching (CPM 99)*, 1999.
[18] G. Navarro and J. Tarhio, "Boyer-moore string matching over ziv-lempel compressed text," in *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pp. 166 – 180, 2000.
[19] S. Klein and D. Shapira, "A new compression method for compressed matching," in *Proceedings of data compression conference DCC-2000, Snowbird, Utah*, pp. 400–409, 2000.
[20] M. Farach and M. Thorup, "String matching in lempel-ziv compressed strings," in *27th annual ACM symposium on the theory of computing*, pp. 703–712, 1995.

[5]One direction is to use one more status type. Recall that we have 2 bit per status but we use only 3 statuses.

[21] U. Manber, "A text compression scheme that allows fast searching directly in the compressed file," *ACM Transactions on Information Systems (TOIS)*, pp. 124 – 136, 1997.
[22] M. Takeda, Y. Shibata, T. Matsumoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa, "Speeding up string pattern matching by text compression. the dawn of a new era.," *Transactions of Information Processing Society of Japan*, pp. 370–384, 2001.
[23] N. Ziviani, E. de Moura, G. Navarro, and R. Baeza-Yates, "Compression: A key for next-generation text retrieval systems," *Computer*, 2000.
[24] "Modsecurity." http://www.modsecurity.org (accessed on July 2008).
[25] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction To Algorithms, The MIT Press and McGraw-Hill Book Company, 2001*.

Appendix

Correctness of ACCH algorithm

In this section we prove theorem 1, the correctness of ACCH algorithm. The proof relies heavily on the following characteristics of AC FSM: The state of the AC FSM after reading as input a series of bytes is correlated to the pattern that has the longest prefix which is a suffix of the input (see Lemma 2). An important outcome of this lemma, is that if the depth of state s in the AC FSM is equal to k then only the last k bytes of the uncompressed traffic are relevant to the state (see Corollary 3).

Formally, let $U_1 \cdots U_j$ be the uncompressed bytes of the input traffic after scanning j bytes, and let P be a set of patterns, P_i a pattern such as $P_i \in P$ where $P_i = P_{i_1} \cdots P_{i_n}$. Let s be the state of AC FSM after scanning all bytes up to U_j.

Lemma 2: For any $P_i \in P$, the length of the longest prefix of any pattern in P which is a suffix of $U_1 \cdots U_j$ is equal to $depth(s)$.

Proof: See [25].

Let the depth of AC FSM state after scanning byte U_j be k. Then,

Corollary 3: The state of AC FSM after scanning $U_1 \cdots U_j$ is equal to the state as if the input traffic to AC FSM is $U_{j-k+1} \cdots U_j$.

The following Lemma 4, is the heart of the proof of the Theorem 1.

Let $Trf = Trf_1 \cdots Trf_N$ be the input, the compressed traffic (after Huffman decompression) and let $U_{j_1} \cdots U_{j_n}$ be the uncompressed string of Trf_j. If Trf_j is a byte then $n = 1$. Let U_{j_i} be a byte in the decompressed traffic - we define $ACCH\_status_{U_{j_i}}$ (and $Naive\_status_{U_{j_i}}$) to be the status of U_{j_i} according to $ACCH$ algorithm (and $Naive$ respectively) after scanning all input bytes before it. Similarly, we define $ACCH\_state_{U_{j_i}}$ ($Naive\_state_{U_{j_i}}$) to be the state we reach at the AC FSM.

Lemma 4: For every Trf_m, where $m \leq N$ the following three claims hold:

1) For every U_{m_i} ($m_i < m_n$), **if** $Naive\_status_{U_{m_i}} = Check$ then $ACCH\_status_{U_{m_i}} = Check$.
2) For every U_{m_i} ($m_i < m_n$), **Iff** $Naive\_status_{U_{m_i}} = Match$ then $ACCH\_status_{U_{m_i}} = Match$ (Note, the two directions of the statement hold).
3) $Naive\_state_{U_{m_n}} = ACCH\_state_{U_{m_n}}$ I.e., the state after decompressing the m compressed input (pointer or a byte) is the same.

Proof: Note that theorem 1 is a straight outcome from claims 2 and 3. The proof is by induction. For $m = 1$, Trf_1 is the first byte (cannot be a pointer) in the traffic and decompresses to U_{1_1}. Thus the induction assumption follows, since the two algorithms have started from the same start state and had the same input.

The induction step, we assume the claim holds until $m-1$, we proof for m.

In the case Trf_m is a byte (and not a pointer), it is easy to see that all three claims hold, since after scanning Trf_{m-1} the two algorithms are in the same state of the AC FSM (from Claim 3) and receive the same input, hence will reach the same state, and will have the same status.

The delict case, is when Trf_m is a pointer. We will go over by induction on the uncompressed form of the pointer i.e., on $U_{m_1} \cdots U_{m_n}$. For readability let $Pt_1 \ldots Pt_{len}$ be equal to $U_{m_1} \cdots U_{m_n}$, where $len = m_n$ We will prove the claims on each pointer part separately: left boundary, internal and right boundary.

Left Boundary: Let us look on the first part, where we search if there exists a pattern in the left boundary of the pointer (lines 17-21). The claim follows, based on Claim 3 and on the induction assumption, the state of the two algorithms is the same at the point before we start scanning the pointer. The two algorithms start at the same state, receive the same input and hence will have the same state and status for all the input bytes until the Pt_j byte where $Depth(ACCH\_state_{Pt_j}) = Depth(Naive\_state_{Pt_j}) \leq j$. From Corollary 3 at this point the pattern prefix that the $Naive$ and $ACCH$ algorithms try to match is completely contained within referred bytes. From this point, we can assume that any pattern prefix that either $Naive$ or $ACCH$ locate is fully contained within referred bytes (i.e., internal area case).

Internal Area: The proof here is divided to two parts, proof on the skipped bytes (lines 28-35) and proof on the scanned bytes (lines 36-39).

Skipped Bytes: In lines 28-29 we skip scanning some bytes, and update the status of those bytes according to the status of the referred bytes. Since we are not in the end of the pointer, we need only to prove the first two claims. *Claim 1:* we need to prove that for every Pt_i ($j < i < p - CDepth$), if $Naive\_status_{Pt_i}$ = $Check$ then $ACCH\_status_{Pt_i} = Check$. Our proof proceeds by reductio ad absurdum. Let us look on the first index i where the claim does not hold. Hence $Naive\_status_{Pt_i} = Check$ and $ACCH\_status_{Pt_i} = Uncheck$ (we do not need to prove here that it cannot be $Match$ since this is straight outcome from Claim 2). Since $Naive\_status_{Pt_i} = Check$ the depth of the state after scanning the Pt_i byte $\geq CDepth$, however the same pattern exists fully also within referred bytes (see end of left boundary proof). The status of the $Naive$ algorithm at the corresponding referred byte in the sliding window is correlated to the longest possible prefix of a pattern that is a suffix of the text scanned so far (see Lemma 2) and hence the status of the referred byte is also $Check$ (it can be only in state with higher or equal depth). From the inductive assumption of Claim 1 the status of the referred byte is also $Check$ for the $ACCH$ algorithm. Since the $ACCH\_status_{Pt_i}$ is set according to the status of referred byte in the sliding window (line 29) (i.e., $Check$) we receive contradiction.

Claim 2: Here we need to prove that for every Pt_i ($j < i < p - CDepth$), iff $Naive\_status_{Pt_i}$ = $Match$ then $ACCH\_status_{Pt_i} = Match$. We first show that there is no $Match$ status in the skipped byte of $ACCH$, and hence the direction that if $ACCH\_status_{Pt_i} = Match$ then $Naive\_status_{Pt_i} = Match$, is proven straight forward from this fact. From the definition of p there is no $Match$ in the corresponding referred bytes. Since the $ACCH\_status$ in the referred bytes in index $j \cdots p - CDepth$ is the same as the status in the sliding window (line 29) there is no $Match$ in $ACCH\_status_{Pt_i}$. The direction that if $Naive\_status_{Pt_i} = Match$ then $ACCH\_status_{Pt_i} = Match$ is proven in a similar way to the proof of Claim 1.

Scanned Bytes: Here we will prove the claims for the bytes in the scanned area (i.e., lines 36-39). The statuses of the $CDepth - 1$ bytes before p are maintained from the sliding window, the same way as in the skipped bytes area. Therefore Claims 1 and 2 hold for those bytes too. We continue the prove from position p.

Claims 1 and 2: For l where $p \leq l \leq k$ we would prove that $Naive\_state_{Pt_l}$ and $ACCH\_state_{Pt_l}$ are equal and hence claims 1,2 follow. Since we are after the left boundary area, we are ensured that all patterns are within pointer boundaries and were copied completely from the referred bytes and hence have the same status as in the referred bytes. Let us look at point p, (recall p is the maximal index before k where the status of the corresponding referred byte at $ACCH$ is $Uncheck$): it is easy to prove that $Naive\_status_{Pt_p}$ is equal to the status of its referred byte in the sliding window (proof by reductio ad absurdum). From the induction assumption Claim 1, the status of this referred byte is the same as of the status according to $ACCH$ algorithm. From definition of p, the status is $Uncheck$ and hence the state of the referred byte according to both algorithms is with distance smaller than $CDepth$. Hence to the AC state relevant only the last $CDepth$ bytes ($Pt_p - CDepth + 1 \ldots Pt_p$) (from Corollary 3). Since we reset the state of FSM in the ACCH (see algorithm line 30), we can prove in a similar way that only the last $CDepth$ bytes are relevant to the ACCH state. Hence both algorithms are in the same state at point p. Therefore from this point $Naive\_state_{Pt_l} = ACCH\_state_{Pt_l}$, for any l where $p \leq l \leq k$.

Right Boundary: Note that in the previous proof on the scanned area, we use only the fact that until k (not including k) there is no referred byte with $Match$ status. Hence the claim also follows for the case where $k = len - 1$. In this case we need also to prove Claim 3, which we prove by showing that for l, $p \leq l \leq k$ (i.e., including k) the states of the l bytes are the same in both $Naive$, and $ACCH$ algorithm. ∎