Accelerating Multi-Pattern Matching on Compressed HTTP Traffic

Anat Bremler-Barr, *Member, IEEE,* and Yaron Koral

***Abstract*—Current security tools, using 'signature-based' detection, do not handle compressed traffic, whose market-share is constantly increasing. This paper focus on compressed HTTP traffic. HTTP uses GZIP compression and requires some kind of decompression phase before performing a string-matching.**

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. By analyzing real HTTP traffic and real web application firewall signatures, we show that up to 84% of the data can be skipped in its scan. Surprisingly, we show that it is faster to perform pattern matching on the compressed data, with the penalty of decompression, than on regular traffic. As far as we know, we are the first paper that analyzes the problem of 'on-the-fly' multi-pattern matching on compressed HTTP traffic and suggest a solution.

***Index Terms*—Compressed HTTP, Computer security, Intrusion detection, Pattern matching**

I. Introduction

SECURITY tools, such as Network Intrusion Detection System (NIDS) or Web Application Firewall (WAF) uses *signature-based detection* techniques to identify malicious activities. Today, the performance of the security tools is dominated by the speed of the underlying string-matching algorithms that detect these signatures [1].

HTTP compression, also known as *content encoding*, is a publicly-available method to compress textual content transferred from web servers to browsers. Most popular sites and applications such as Yahoo!, Google, MSN, YouTube and Facebook use HTTP compression; as of 2007, over 27% of the Fortune 1000 companies web sites used HTTP compression and the trend is increasing [2]. Moreover, this standard method of delivering compressed content is built into HTTP 1.1, and thus supported by most modern browsers. On average, content encoding saves around 75% of transmitted text files (HTML, CSS, and JavaScript) [3].

Multi-pattern matching on compressed traffic requires two time-consuming phases, namely *traffic decompression* and *pattern matching*. Therefore, most current security tools either ignore scanning compressed traffic, which may be the cause of security holes or disable the option for compressed traffic by re-writing the 'client-to' HTTP header to indicates that compression is not supported by the client's browser thus decreasing the overall performance and bandwidth. Few security tools handle HTTP compressed traffic by decompressing the entire page on the proxy and performing signature scan on the decompressed page before forwarding it to the client. That option is not applicable for security tools that operate at a high speed or when introducing additional delay is not an option.

In this paper we present a novel algorithm, *Aho-Corasick-based algorithm on Compressed HTTP (ACCH)*. ACCH decompresses the traffic and then uses the data from the decompression phase to accelerate the pattern matching. Specifically, GZIP compression algorithm works by eliminating repetitions of strings using back-references (pointers) to the repeated strings. Our key insight is to store information produced by the pattern matching algorithm for the already scanned decompressed traffic, and in the case of a pointer, use this information to understand if it contains a match or one can safely *skip scanning bytes within it*. We show that ACCH can skip up to 84% of the data and boost the performance of the multi-pattern matching algorithm by up to 74%.

Preliminary abstract of this paper was published in the proceedings of IEEE INFOCOM 2009.

II. Background

GZIP algorithm: GZIP [4] is the most commonly used algorithm for compressing HTTP 1.1 traffic. It is based on the DEFLATE [5] algorithm that first compresses the text using LZ77 algorithm [6] and then compresses the output using Huffman coding [7].

LZ77 Compression - LZ77 technique compresses a string which has already appeared in the past (specifically, in a sliding window of the last 32KB of decompressed data) by encoding it with a pair (*distance*, *length*), where *distance* is a number in $[1, 32\,768]$ that indicates the distance in bytes to the repeated string, and *length* is a number in $[3, 258]$ that indicates the length of the string in bytes. For example, the string: 'abcdefabcd', can be compressed to: 'abcdef(6,4)', where $(6, 4)$ implies one should return 6 bytes and copy 4 bytes from that point. Fig. 1 shows example of 'Yahoo!' home page after compression (Stage I - LZ77).

Note that decompression has moderate time consumption, since it reads and copies sequential data blocks, hence relying on spatial locality requiring only few memory references.

Huffman Coding - The Huffman method works on character-by-character basis, transforming each 8-bit character to a variable-size *codeword*; the more frequent the character is, the shorter its corresponding codeword. The codewords are coded such no codeword is a prefix of another so the end of each codeword can be easily determined. *Dictionaries* are provided to facilitate the translation of binary codewords to bytes.

A. Bremler-Barr is with the Efi Arazi School of Computer Science, The Interdisciplinary Center, Herzlia, Israel (email: bremler@idc.ac.il)

Y. Koral is with the Blavatnik School of Computer Sciences Tel-Aviv University, Israel (email:yaronkor@post.tau.ac.il).

Manuscript received March 23, 2010; revised April 18, 2011.

Yahoo Decompressed file:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en-US"><head><meta http-equiv=
"Content-Type" content="text/html; charset=UTF-8">
<script type="text/javascript">
var now=new Date,t1=t2=t3=t4=t5=t6=t7=t8=t9=t10=t11=t12=0,cc='',
ylp='';t1=now.getTime();
</script>
```

(a)

Yahoo LZ77 form:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD{26,6}4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<{20,4} lang="en-US{20,5}head{7,3}meta {73,4}-equiv=
"Content-Type" c{14,6}="text{92,5}; charset=UTF-8{75,4}
script t{50,3}{41,6}java{22,6}{32,3}
var now=new Date,t1=t2=t3=t4=t5=t6=t7=t8=t9=t10{4,3}{32,3}12=0,cc='',
ylp{7,3};{54,3}{70,3}.getTime();
</{100,6}>
```

(b)

Fig. 1. Example of the LZ77 compression on the beginning of the Yahoo home page (a) Original (b) After the LZ77 compression

In the DEFLATE format, Huffman codes both ASCII characters (a.k.a. *literals*) and pointers into codewords using two dictionaries, one for the literals and pointer lengths and the other for the pointer distances. Huffman may use either fixed or dynamic dictionaries. The use of dynamic dictionaries gains better compression ratio. The Huffman dictionaries for the two alphabets appear in the block immediately after the header bits and before the actual compressed data.

A common implementation of Huffman decoding (cf. zlib [8]) uses two levels of lookup tables. The first level stores all codewords of length less than 9 bits in a table of 2^9 entries that represents all possible inputs; each entry holds the symbol value and its actual length. If a symbol exceeds 9 bits, there is an additional reference to a second lookup table. Thus, on most of the cases, decoding a symbol requires only one memory reference, while for the less frequent symbols it requires two.

Multi-pattern matching: Pattern-matching has been a topic of intensive research resulting in several approaches; the two fundamental approaches are based on Aho-Corasick (AC) [9] and the Boyer-Moore [10] algorithms. In this paper, we illustrate our technique using the AC algorithm.

The basic AC algorithm constructs a *Deterministic Finite Automaton* (DFA) for detecting all occurrences of given patterns by processing the input in a single pass. The input is inspected symbol by symbol (usually each symbol is a byte), such that each symbol results in a state transition. Thus, AC algorithm has deterministic performance, which does not depend on the input, and therefore is not vulnerable to various attacks, making it very attractive to NIDS systems.

Fig. 2 depicts the DFA for the patterns set: 'abcd', 'nba', 'nbc'. The resulting automaton has 9 states (corresponding to the nodes of the graph). Each arrow indicates a DFA transition made by a single byte scan. The label of the destination state indicates the scanned byte. If there is no adequate destination state for the scanned byte, the next state is set to root. For readability, transitions to root were omitted.

Note that this common encoding requires a large matrix of size $|\Sigma| \cdot |S|$ where Σ is the set of ASCII symbols and S in the number of states) with one entry per DFA edge. In the typical case, the number of edges, and thus the number of entries, is $256|S|$. For example, Snort patterns [11] used in the experimental results section, require 16.2MB for 1202 patterns that translate into 16 649 states. There are many compression algorithms for the DFA (for example [12]–[16]) but most of them are based on hardware solutions.

At the bottom line, DFAs require significant amount of memory, therefore they are usually maintained in main memory and characterized by random rather than consecutive accesses to memory.

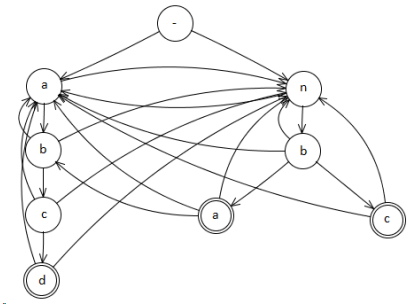


Fig. 2. Aho Corasick DFA for patterns: 'abcd', 'nba', 'nbc'.

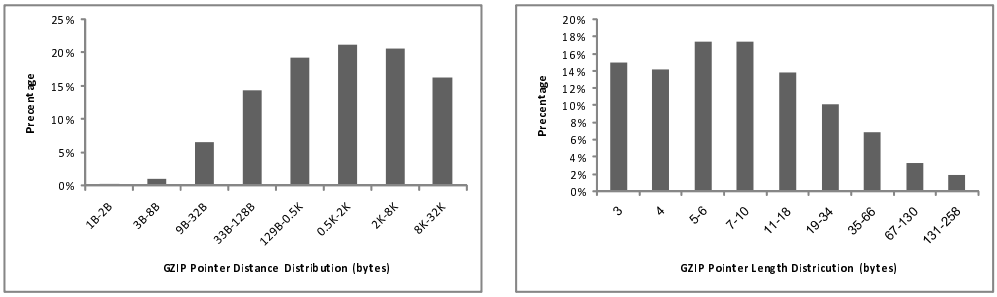
III. The challenges in performing multi-pattern matching on compressed HTTP traffic

This section provides an overview of the obstacles in performing multi-pattern matching on compressed HTTP traffic. Note that dealing with web traffic is significantly more challenging than its offline counterpart, since the compression method cannot be chosen or modified. Second, pattern matching should be performed '**on-the-fly**'.

We note that there is no apparent "easy" way to perform multi-pattern matching over compressed traffic without decompressing the data in some way. This is mainly because *LZ77* is an *adaptive* compression; namely, the text represented by each symbol is determined dynamically by the data. As a result, the same substring is encoded differently depending on its location within the text. Thus, decoding the pattern is futile. For example the pattern 'abcd' can be expressed in the compressed data by $abc*^j\ (j+3,3)d$ for all possible $j < 32\,765$. On the other hand, Huffman encoding, is non-adaptive within a given text and the same symbol will always be encoded to the same bit sequence. However, the combination of the two algorithms is adaptive.

The naïve way of performing multi-pattern matching on the traffic in real time, is by 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.



(a) (b)

Fig. 3. Distribution of the following pointer characteristics on the real life data set of Section VII (a) distance of a pointer (b) length of a pointer.

Algorithm 1 Naïve: Decompression with Aho-Corasick

Trf - the input, compressed traffic (after Huffman decompression).
$SlideWin_{0...32KB}$ - LZ77 sliding window, contains last 32KB of decompressed data.
$SlideWin_j$ - Represents the j^{th} entry prior to current position.
$DFA(state, byte)$ - performs single DFA transition and returns the next state. $startStateDFA$ is the initial DFA state.
$Match(state)$ - stores information about the matched pattern if $state$ is a "match state", otherwise it is NULL.

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

7: procedure GZIPDecompressPlusAC(Trf_1 ... Trf_n)
8: state=startStateDFA
9: for i=1 to n do
10:    if Trf_i is pointer (dist, len) then
11:        for j = 0 to len-1 do
12:            state = scanAC(state, SlideWin_{dist-j})
13:        end for
14:        SlideWin_{0...len-1} = SlideWin_{dist...dist-len-1}
15:    else
16:        state = scanAC(state, Trf_i)
17:        SlideWin_0 = Trf_i
18:    end if
19: end for
```

4) Perform multi-pattern matching on the decompressed traffic.

The challenges of the multi-pattern 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 32KB sliding window for each HTTP session. Note that this requirement is difficult to avoid, since the back-reference pointer can refer to any point within the sliding window and the pointers may be recursive unlimitedly (i.e., pointer may point to area with a pointer). Fig. 3(a) shows that indeed the distribution of pointers on real-life data set (see Section VII for details on the data set) is spread across the entire window. On the other hand, pattern matching of non-compressed traffic requires storing only one or two packets (to handle cross-packet data), where the maximum size of TCP packet is 1.5KB. Hence, dealing with compressed traffic poses a higher memory requirement by a factor of 10. Thus, mid-range firewall, that handles 30K concurrent sessions, requires 1GB memory while a high-end firewall with 300K concurrent sessions requires 10GB. This memory requirement has implication on not only the price and feasibility of the architecture but also on the capability to perform caching. The space requirement is not in the focus of this paper. Still, recent work by Afek et al. [17] has shown techniques that circumvent that problem and drastically reduce the space requirement by over 80%, with only a slight increase in time. It has also shown a method to combine that technique with ACCH that achieves improvements of almost 80% in space and above 40% in the time requirement for the overall DPI processing of compressed web traffic.
Time - Recall that pattern matching is a dominant factor in the performance of security tools [1], while performing decompression further increases the overall time penalty. Therefore, security tools tend to ignore compressed traffic. This work focus on reducing the time requirement by using the information gathered by compression phase.

We note that pattern matching with the AC algorithm requires significantly more time than decompression, since decompression is based on consecutive memory reading from the sliding window hence, has low read-per-byte cost. On the other hand, the AC algorithm employs a very large DFA that is accessed with random memory reads, that typically does not fit in cache thus requiring main memory accesses.

Appendix A introduces a model that compares the time requirements of the decompression and the AC algorithm. Experiments on real data show that decompression takes only a negligible 3.5% of the time it takes to run the AC algorithm. For that reason, we focus on *improving AC performance*. We show that we can reduce the AC time by skipping more than 70% of the DFA scans and hence reduce the total time requirement for handling pattern matching in compressed traffic by more than 60%.

IV. Related Work

The problem of pattern matching on compressed data has received attention in the context of the Lempel-Ziv compression family [18]–[21]. However, the LZW/LZ78 are more attractive and simple for pattern matching than LZ77. 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. Klein and Shapira [22] suggest modification to the LZ77 compression algorithm to make the task of the matching easier in files. However, the suggestion is not implemented in today's HTTP.

[23], [24] are the only papers we are aware of that deal with pattern matching over LZ77. However, in those papers the algorithms are for single pattern and require two passes over the compressed text (file), which is not applicable for network domains that requires 'on-the-fly' processing.

One outcome of this paper is the surprising conclusion that pattern matching on compressed HTTP traffic, with the overhead of decompression, is faster than pattern matching on regular traffic. We note that other works with the context of pattern matching in compressed data such as [25]–[27] has shown a similar conclusion, stating that compressing a file once and then performing pattern matching on the compressed file accelerates 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)* (detailed pseudo code in Algorithm 2). The algorithm performs multi-pattern matching over compressed input and provides the exact location of all occurrences of any pattern from the given pattern-set within the uncompressed form of the input data stream. As recalled, HTTP uses GZIP that compresses data with pointers to past occurrences of strings. ACCH operates on the uncompressed data after decompression but uses the information about the pointer structure within the compressed data. We start with some definitions. Let *string* be a sequence of consecutive bytes within the uncompressed data. Recall that *pointer* is a recurrence of a *string* within the uncompressed data represented by a pair (*distance*, *length*). Let the string that the *pointer* points to be denoted as *referred string*. Hence *distance* indicates the number of bytes between the *referred string* start to the *pointer* start within the uncompressed data and *length* indicates the number of bytes of the *referred string* in the uncompressed data which are equal to the number of bytes in the uncompressed data represented by the *pointer* (see Figure 4). Therefore if the *referred string* starts at position k within the uncompressed data then the following byte sequences, representing the *referred string* and *pointer* respectively, are equal: $\{b_k, \ldots, b_{k+length-1}\} = \{b_{k+distance}, \ldots, b_{k+distance+length-1}\}$.

The algorithm operates the same as the basic AC algorithm as long as the input data contains literals (Lines 49-53), and updates a status vector that would be explained on the sequel. The basic observation is that if the *referred string* does not completely contain matched patterns then the *pointer* also contains none. Following this observation implies that the algorithm may skip scanning bytes in the uncompressed data where the *pointer* occurs. Hence, the algorithm handles three special cases where the first two refer to examining whether the *pointer* is part of a pattern and the third refers to the case where a match occurred within the *referred string*. In the first case, the pattern starts prior to the *pointer* and only its suffix is in the *pointer*. In the second case the *pointer* contains a pattern prefix and its remaining bytes occur after (see Example 1 in Fig. 5). In order to detect those patterns we need to scan few bytes within the *pointer* starting and ending points denoted by pointer *left boundary* and *right boundary* scan areas respectively (as in Fig. 4). If no matches occurred within the *referred string* the algorithm skips the remaining bytes between the *pointer* boundaries denoted by *internal area*. Note that *left boundary*, *right boundary* and *internal area* are properties determined by the ACCH algorithm as opposed to the *pointer* and its *referred string*. In the third case a pattern ends within the *referred string*. The algorithm has to figure out whether the whole pattern is contained within the *referred string* or only its suffix (see Example 2 in Fig. 5). We now describe how the algorithm handles these three cases:

Left Boundary (first case): In this case the algorithm determines whether a pattern ends within the *pointer* left boundary. We use the state *depth* parameter (namely, the number of edges on the shortest simple path from the state and the DFA root). The *depth* of a state has a property that it indicates the longest prefix of any pattern within the pattern-set [28] at the current scan location. Hence, the algorithm should continue scanning the *pointer* bytes (in the uncompressed data) as long as the number of bytes scanned within the *pointer* is smaller than the current DFA state's *depth* (Lines 12–18). Note that if the current state *depth*, prior *pointer* area is 0, the algorithm moves directly to the *internal area* case without scanning a single byte (as shown in Examples 2 and 3 in Figure 5).

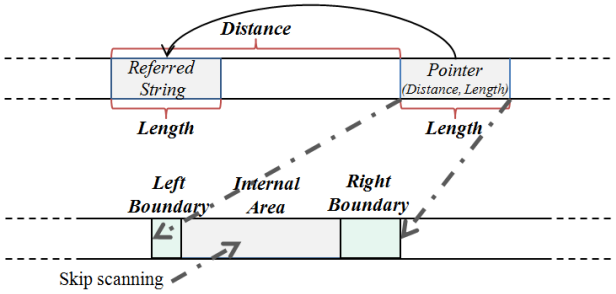


Fig. 4. Illustration of *pointer* and its *referred string* with the *left boundary* and *right boundary* scan areas.

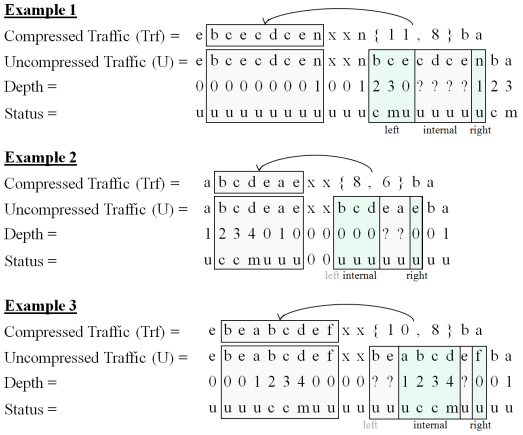


Fig. 5. ACCH run for *CDepth*=2 using the DFA from Fig. 2. Gray rectangles indicate *pointer* and its *referred string*. Light green rectangles indicate bytes which the algorithm scan. Question marks indicate that the corresponding bytes were skipped therefore their depth is unknown to the algorithm.

Algorithm 2 Compressed HTTP based Aho-Corasick

Definitions are as in algorithm 1, with the following additions:
$byteInfo$ - is a record that contains a pair of variables: b - byte value; $status$ - byte status.
$SlideWin$ - A cyclic array where each entry is of type $byteInfo$.
$SlideWin_j$ - Represents the j^{th} entry prior to current position.
$Depth(state)$ - a function that returns the depth of the state in the DFA.
$CDepth$ - the constant parameter of ACCH algorithm.

1: **(**$status$**,**$nextState$**) = function scanAC(**$state$**,** $byte$**)**
2: $nextState$ = DFA($state$, $byte$)
3: **if** $Match(nextState) \neq NULL$ **then**
4: act according to $Match(nextState)$
5: $status$ = Match
6: **else**
7: **if** $Depth(nextState) \geq CDepth$ **then** $status$ = Check
8: **else** $status$ = Uncheck
9: **end if**
10: **end if**
11: return ($status$,$nextState$)

12: **(**$nextPos$**,** $nextState$**) = function scanLeft(**$state$**,** $curPtrInfo$**)**
13: $curPos$=0
14: **while** $(Depth(state) > curPos) and (curPos < len)$ **do**
15: ($status$, $state$) = scanAC($state$, $SlideWin_{dist-curPos}.b$)
16: $curPtrInfo[curPos].status = status$
17: $curPos$++
18: **end while**
19: return ($curPos$, $state$)

20: **(**$nextPos$**,** $nextState$**) = function scanSegment(**$state$**,** $start$**,** end**,** $curPtrInfo$**)**
21: Find the maximal $unchkPos$, $start \leq unchkPos \leq end$ such $SlideWin_{dist-unchkPos}.status = Uncheck$
22: If no such $unchkPos$ exists $unchckPos = start$ ▷ Continue from last scanned byte without skip
23: **if** $start < (unchkPos - CDepth + 2)$ **then** ▷ Skip and update window
24: $curPtrInfo[start \ldots (unchkPos - CDepth + 2)].status = SlideWin_{dist-start \ldots (dist-(unchkPos-CDepth+2))}.status$
25: $state = startStateDFA$
26: **for** $curPos = (unchkPos - CDepth + 2)$ to $(unchkPos)$ **do** ▷ Scan bytes and copy status from $SlideWin$
27: ($state$,$status$)=scanAC($state$,$SlideWin_{dist-curPos}.b$)
28: $curPtrInfo[curPos].status = SlideWin_{dist-curPos}.status$
29: **end for**
30: **end if**
31: **for** $curPos = unchkPos + 1$ to end **do** ▷ Internal or Right Boundary scan
32: ($state$, $status$) = scanAC($state$, $SlideWin_{dist-curPos}.b$)
33: $curPtrInfo[curPos].status = status$
34: **end for**
35: return ($curPos$+1,$state$)

36: **procedure ACCH(**$Trf_1 \ldots Trf_n$**)**
37: **for** i=1 to n **do**
38: **if** Trf_i is pointer ($dist$,len) **then**
39: $curPtrInfo[0 \ldots len-1].b = SlideWin_{dist \ldots dist-len}.b$
▷ $curPtrInfo$ - an array of $byteInfo$ entries, used to maintain pointer status information during pointer scan
40: ($state$, $curPos$) = scanLeft($state$,$curPtrInfo$)
41: **while** $curPos < len$ **do** ▷ Check $Match$es within internal area
42: Find the minimal $matchPos$, $curPos \leq matchPos < len$ such $SlideWin_{dist-matchPos}.status$ = Match
43: **if** no such $matchPos$ exist **then** ▷ Case of Right Boundary Segment
44: $curPos$=scanSegment($state$, $curPos$, len-1, $curPtrInfo$)
45: **else** $curPos$=scanSegment($state$, $curPos$, $matchPos$, $curPtrInfo$) ▷ Case of Match Segment
46: **end if**
47: **end while**
48: $slideWin_{len-1 \ldots 0} = curPtrInfo[0 \ldots len-1]$
49: **else** ▷ Trf_i is not a pointer, performing simple byte scan
50: $SlideWin_0.b = Trf_i$
51: ($state$,$status$)=$scanAC(state, SlideWin_{dist-i}.b)$
52: $SlideWin_0.status = status$
53: **end if**
54: **end for**

Right Boundary (second case): In this case the algorithm determines whether a pattern starts within the *pointer* suffix. As opposed to the *left boundary* case, we do not have a current DFA state since the bytes prior to the *pointer*'s *right boundary* were skipped. One may suggest that the DFA state can be maintained from prior scans of the *referred string*. That would require maintaining all previous scan states. Those states are determined by DFA scans. But since ACCH skips scanning bytes, it has no way of determining those bytes exact state. Another option is to use the *depth* property as mentioned in the *left boundary* case. Since the *depth* of a state expresses the longest prefix of any pattern within the pattern-set, it indicates the maximal number of bytes that need to be scanned within pointer *right boundary* in order to detect a prefix. Still, maintaining the exact *depth* is also complicated due to the presence of skipped bytes. Therefore we use an approximation to the *depth* parameter which is easy to maintain and also requires less space. The approximation indicates whether the depth of the corresponding byte is below some constant *CDepth*, a parameter of our algorithm. In such a case this byte is marked with a status of *Uncheck*; otherwise the byte is marked *Check*. Marking a byte with a status is done by storing a bit for each byte within the sliding window. In the case where a byte was scanned, the status is obtained according to the depth of the state the DFA reached. The case where a byte was skipped is discussed in the sequel.

Having the above status information, the algorithm locates the last occurrence of an *Uncheck* status position within the *referred string*. Let *unchkPos* be the corresponding position within the *pointer*. In order to determine safely the *right boundary* area, the scan is resumed from *unchkPos* $-$ $CDepth + 2$ bytes prior *pointer* end and the DFA state is set to *start state* (see Function *scanSegment* Lines 20–35).

Note that a byte with an *Uncheck* status provides a location to restart scanning from. Hence every byte with *Uncheck* status raises the chance for skipping more bytes. As noted in [12], [13] and shown in our experimental results section, most of the time the DFA uses states of low depth, hence in most cases the status would be *Uncheck*, therefore we might skip scanning most of the bytes.

The challenging task is to maintain a correct status for the skipped bytes. This is done by simply copying the status information from the corresponding bytes within the *referred string*. Since we finished handling the *left boundary* case, we are certain that if the skipped bytes would have been scanned, their depths would not have been higher than the depths of the corresponding bytes within the *referred string*. Hence, copying statuses might over estimate the depth of a byte thus assigning it a *Check* status rather than an *Uncheck* (but not vice-versa) in the case of skipped bytes. Still, a mistake in that direction does not cause misdetection. It only causes skipping less bytes.

Internal area (third case): (Lines 41–47) Up to this point we assumed that no match occurred within the *referred string*. Information about previous matches is crucial for the algorithm in order to determine whether the previously matched pattern was referred to entirely by the *pointer* or only its suffix. First we introduce a third status (along with *Uncheck* and *Check*) called *Match*. This status is stored using additional bit for each byte within the sliding window. A byte with a *Match* status means that a pattern (or more) ends at its location. Let *matchPos* be the position within the pointer, corresponding to the position of the $Match$ status within the *referred string*. Using the status information, we could detect whether an entire pattern was referred to by the *pointer*, by scanning few bytes prior the *matchPos* in the same manner as in the case of the *right boundary*. We locate *unchkPos*, the last *Uncheck* status prior the byte with the *Match* status, and start scanning *unchkPos* $-$ $CDepth + 2$ bytes prior to *matchPos*. Function *scanSegment* is also used for this case (Lines 20–35).

In the next theorem we prove the correctness of the algorithm. Let P be a finite set of patterns, Trf the compressed traffic and U the decompressed 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 Appendix B. The proof relies on the validity of AC algorithm. Therefore, we perform pattern matching on the compressed traffic twice, once with the naïve algorithm (decompression + AC that scans all bytes), denoted as the Naïve algorithm, and another time with $ACCH$. The two algorithms use the same DFA, 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 Naïve 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 Naïve 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 Naïve detects and theorem follows. 3) Both algorithms, $ACCH$ and Naïve reach exactly the same DFA state after scanning of any single byte or after scanning a decompressed pointer entirely. This is not true within a pointer, since $ACCH$ may skip scanning some of the bytes. The proof relies heavily on the characteristics of AC DFA. Based on Claim 2 from Lemma 4 we show the validity of Theorem 1. ∎

VI. ACCH Optimizations

In this section we present two techniques that exploit the characteristics of ACCH further to achieve even better performance. ACCH algorithm rescans at least $CDepth - 1$ bytes at every *right boundary* or a byte marked as *Match*. This implies that when the data contains many "matches", ACCH performance deteriorates dramatically. Optimization I fits for data with many "matches". It eliminates the need for rescanning prior to *Match* marks. Optimization II refers to another aspect of ACCH. Note that the algorithm uses only three statuses out of four possible, using 2 bits. Optimization II uses a fourth status to achieve better performance.

A. Optimization I - Eliminate Byte Scans Within Internal Area

Optimization I (described by Algorithm 3) focus on the *internal area* (third case). We use a hash table denoted as *Match*

Algorithm 3 ACCH - Optimization I

$absPosition$ - Absolute position from the beginning of data.
After line 38: $absPosition$ += len
After line 49: $absPosition$ ++

$MatchTable$ - A hash table, where each entry represents a Match. The key is the Match $absPosition$ and the value is a list of patterns that where located at that position.

Function **scanAC** - a new line is added after line 4:
add patterns in $Match(state)$ to $MatchTable(absPosition)$

Procedure **ACCH** - instead of the while loop, lines (41-47):
handleInternalMatches($state$, $curPos$, len-1)
scanSegment($state$, $curPos$,len-1)

Function **scanSegment** should ignore Matches found by **scanAC** since all matches within pointer are located by functions **scanLeft** and **handleInternalMatches**.

```
1: function handleInternalMatches(start, end)
2: for curPos = start to (end) do
3:     if SlideWin_{dist-curPos}.status = Match then
4:         if MatchTable(curPos) contains patterns
                   shorter or equal to curPos then
5:             add those patterns to MatchTable(absPosition)
6:             curPtrInfo[curPos].status = Match
7:         else curPtrInfo[curPos].status = Check
8:         end if
9:     else
10:        curPtrInfo[curPos].status
                   = SlideWin_{dist-curPos}.status
11:    end if
12: end for
```

Table. Whenever $scanAC$ reaches a "match state", all patterns that were found by that state are stored in the $MatchTable$. The position of the byte within the uncompressed data is used as a key to that entry in the hash table.

Algorithm 3 changes the *internal area* case (see Lines 1–12) such as for each *Match* found within the *referred string*, the list of its related patterns in $MatchTable$ is extracted. Each pattern in that list with length not longer than the index of the "match status" within the pointer, was referred-to entirely by the *pointer* and therefore is contained within it. All those patterns are added to the $MatchTable$ and the status is marked as *Match* (see Lines 5–6).

The case where all patterns are longer than the index of *Match* indicates that only the suffix of them was referred by the *pointer*, therefore no match occurs at that position within *pointer*. The status of that position is safely set to *Check* since it cannot be maintained from the referred bytes (Line 7), and an error in that direction does not harm algorithm correctness (as discussed in Section V). All status bits of skipped bytes within internal area other than *Match*, are copied from the *referred string* (Line 10).

This optimization gains significant speed improvement as shown in the experimental results section. The downside is the additional data structure $MatchTable$ that has to be maintained.

B. Optimization II - The CDepth Parameter

ACCH, as described in Algorithm 2, uses three possible status types for each byte: $Check$, $Uncheck$ and $Match$. Those types are maintained within the sliding window using two bits. Since two bits represent four different values, additional status can be represented by those bits without using extra space.

Algorithm 4 uses an additional status in order to maintain a more precise estimation about the depth of each byte in the sliding window. We define two constant parameters: $CDepth1$ and $CDepth2$ instead of one, where $CDepth1 < CDepth2$. Status of bytes other than $Match$ is determined in the following way: if depth is smaller than $CDepth1$ then the status is $Uncheck1$. If depth is smaller than $CDepth2$ but not from $CDepth1$ then the status is $Uncheck2$. Otherwise the status is $Check$.

Algorithm 4 ACCH - Optimization II

$CDepth1$, $CDepth2$ - Instead of one constant parameter $CDepth$, we maintain two, where $CDepth1 < CDepth2$

Function **scanAC** - line 8 changes to lines:
else if $Depth(state) \leq CDepth1$ **then** status = $Uncheck1$
else status = $Uncheck2$

Function **scanSegment** - line 21: instead of searching for maximal $Uncheck$, it searches for maximal $Uncheck1$ or $Uncheck2$

Function **scanSegment** - lines 23-30: $CDepth$ parameter changes to $CDepth1$ or $CDepth2$ depending on whether the state found on line 21 is $Uncheck1$ or $Uncheck2$ respectively

$CDepth$ parameter determines two factors that influence the number of bytes scanned by the ACCH algorithm: *constant part*, where $scanSegment$ scans $CDepth - 1$ bytes prior to $unchkPos$; *variable part*, where $scanSegment$ scans all bytes from $unchkPos$ to the end of the segment. Hence, there is a tradeoff choosing different $CDepth$ values, between using a higher value of $CDepth$ which leads to more $Uncheck$s (which in turn lowers the *variable* part) or lower value of $CDepth$ that minimizes the *constant* part.

By using two constant depth estimators instead of one, the algorithm gains the advantages of both factors (see experimental results in Section VII). The algorithm sets as much $Uncheck1$ and $Uncheck2$ status types as it would set $Uncheck$ when $CDepth$ from Algorithm 2 equals $CDepth2$. And $scanSegment$ scans the same amount of bytes as it would have if $CDepth$ from Algorithm 2 would be equal to $CDepth1$. Status is maintained as in Algorithm 2, by either copying the status from the *referred string* or determining the status with $scanAC$ according to state depth.

VII. Experimental Results

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

We use two data sets, one of the traffic and the other of the patterns. For the traffic, we use two data sources: traffic

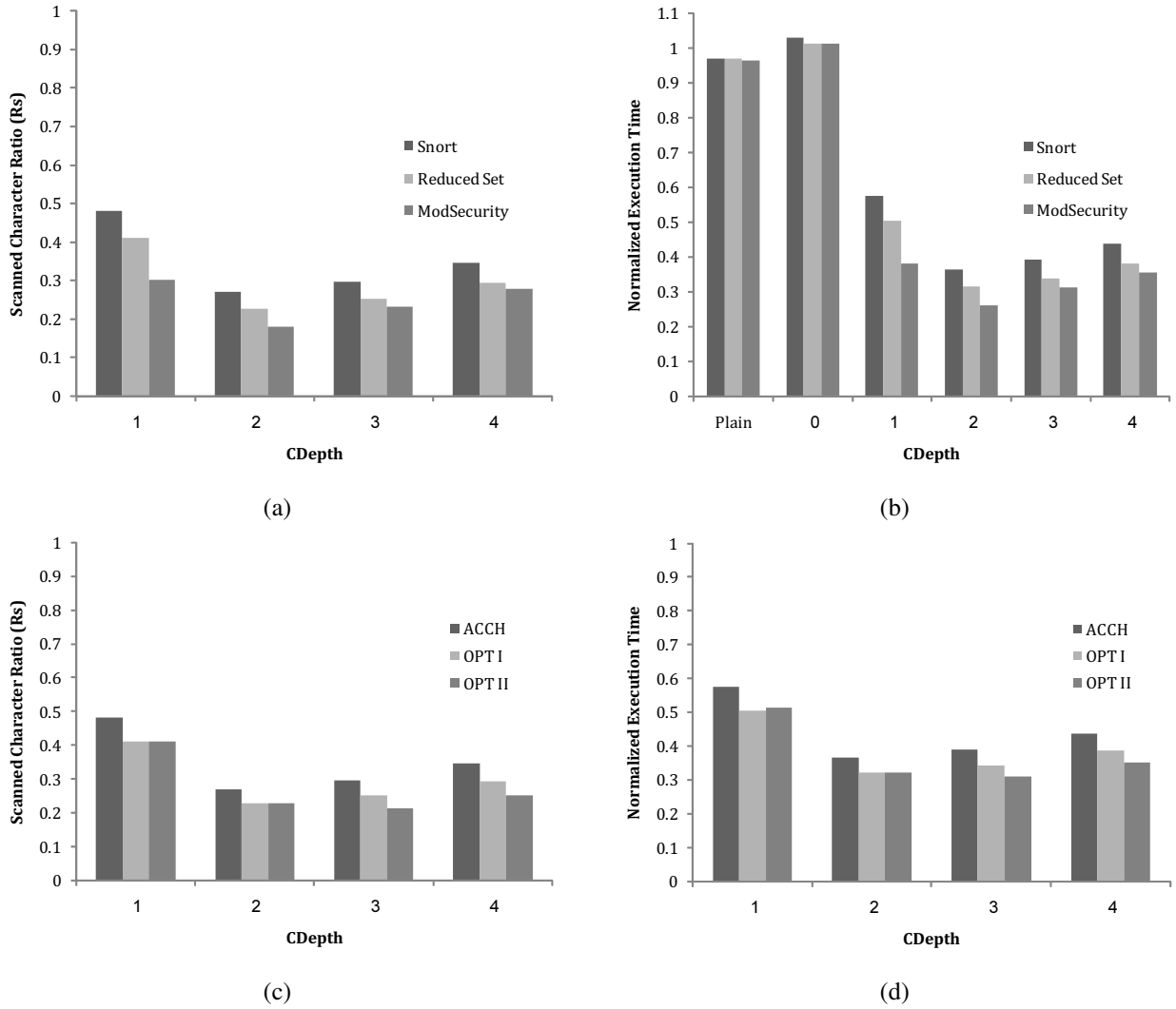


Fig. 6. (a) Scanned character ratio as function of $CDepth$ for ModSecurity, Snort and the Reduced Snort data-set. (b) Performance benefit as oppose to the Scanned character ratio (R_s) as function of $CDepth$ compared to the naïve algorithm (decompression + AC that scans all bytes) performance. "Plain" column, refers performance of running Aho-Corasick on the uncompressed data. (c) Scanned character ratio as function of $CDepth$ for Snort pattern set compared among ACCH with optimizations. (d) Performance benefit as function of $CDepth$ for Snort pattern set compared among ACCH with optimizations.

	Corp. FW	**Alexa**
Num. of HTML pages	23 698	14 078
Uncompressed size	910MB	808MB
Compressed size	181MB	160MB
P_r - Ratio of bytes represented by pointers	0.94	0.921
P_l - Average pointer length	17.8	16.7

TABLE I
PARAMETERS OF BOTH DATA-SETS.

that was captured by a corporate firewall for 15 minutes and a list of the most popular web pages taken from Alexa web site [29], that maintains web traffic metrics and top sites lists. Table I shows the characteristics of both traces. No significant difference was found between the results over those two data sets, therefore we refer to the Alexa data-set parameters.

We use two signature data sets: one of ModSecurity [30], an open source web application firewall, and the other of Snort [11], an open source network intrusion prevention and detection system.

In ModSecurity we chose a signature group that applies to HTTP-response (since only the response is compressed). Patterns are normalized such as they do not contain regular expressions.[1] Total number of patterns is 124.

The Snort data set contains rules which in turn may contain either exact strings or regular expressions. Since this work is restricted to exact string matching techniques and it does not cover regular expression matching, we constructed the signature set from the exact strings only. The extracted signature set contains more than 8K patterns. Most of them are in a binary mode that is not relevant to textual HTML content. Therefore we removed the binary patterns and were left with a subset of 1202 textual patterns. We note that Snort patterns are not meant to be applied on an HTTP-response, hence are less applicable in the domain problem of compressed HTTP traffic. However, we decided to use this signature set since

[1]Regular expressions were extracted into several plain patterns.

most pattern-matching papers refer to it therefore is good for evaluating the effect of the number of patterns and number of matches on our algorithm performance compared to other work in the area. We also note that Snort patterns occur significantly more within the traffic, mainly since Snort has patterns like "st", "id=" or "ver". The data set contains 655 occurrences of ModSecurity patterns and over 14M 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 naïve algorithm as shown later. Fig. 6(a) summarizes R_s as a function of $CDepth$, for Snort and ModSecurity patterns. $CDepth = 2$ shows best performance for both pattern sets. R_s for Snort patterns is **0.27** and for ModSecurity is **0.181**. The tradeoff of choosing different values for $CDepth$, as discussed in Section VI-B, can be shown in the graph.

Snort patterns result in a significant amount of matches within the traffic (near 2% of the total number of bytes are marked with $Match$) resulting in more *Match Segment*s, which in turn cause a significantly higher R_s. In order to check the influence of the amount of matches on R_s we synthesized a *reduced Snort pattern-set* by removing 88 of the most frequent patterns. The data contains $234\,448$ patterns from the reduced pattern-set (instead of 14M). As can be seen from 6(a), removing only 88 patterns had a significant effect on the R_s value (R_s = 0.23 for $CDepth = 2$). Thus, match ratio has stronger influence on R_s value than the number of patterns.

Fig. 6(b) shows the correlation between R_s and the performance benefit compared to the performance of running the naïve algorithm (i.e. decompression + AC). We use Intel Core 2 Duo 2GHZ with 2GB RAM as a platform. The "Plain" column shows the performance ratio of running AC on plain uncompressed data. ACCH with $CDepth = 0$ marks all statuses as $Check$, therefore no scan-skips takes place. It also shows a slight overhead in implementing the algorithm as compared to the naïve algorithm, which is between 4%-8%. It can be explained by the additional memory used for the status bits in the sliding window, which in turn result in more memory references. For $CDepth = 2$, ACCH running over ModSecurity achieved 74% performance improvement and ACCH running over Snort achieved 64%.

The surprising outcome is that the algorithm operates faster on compressed data than on uncompressed, as shown by comparing the "Plain" column in Fig. 6(b) to other columns with $CDepth > 0$.

C. ACCH Optimizations

Fig. 6(c) shows the impact on R_s for both optimizations compared to ACCH in Algorithm 2 for the **Snort** pattern set. For Optimization II we used several combinations of $CDepth1$ and $CDepth2$ values. Fig. 6(c) shows four combinations where $CDepth2 = CDepth1 + 1$. The values of the x-axis refer to $CDepth2$. In the first column $CDepth1 = CDepth2 = 1$. For Optimization I, $CDepth = 2$ is optimal with $R_s =$ **0.229** and for Optimization II $CDepth1 = 2$ and $CDepth2 = 3$ is optimal with $R_s =$ **0.215** as compared to $R_s =$ **0.27** for ACCH in Algorithm 2.

	Description	Snort	ModSecurity
A_l	Average byte number scanned on the *left boundary*	1.077	0.684
A_r	Average byte number scanned on the *right boundary*	1.61	1.159
A_m	Average byte number scanned on *Match Segment*	2.55	4.824
M_r	Ratio of bytes with $Match$ status out of all bytes	0.017	7.73×10^{-7}

TABLE II
PARAMETERS ANALYZED USING REAL-LIFE DATA FOR SNORT AND MODSECURITY. PARAMETERS MEASURED WHILE EXECUTING ACCH WITH $CDepth$ OF VALUE 2.

For **ModSecurity** pattern set, $CDepth = 2$ is optimal for Optimization I and for Optimization II the optimal parameters are $CDepth1 = 1$ and $CDepth2 = 2$. In Optimization I, $R_s =$ **0.181** and in Optimization II, $R_s =$ **0.163**. Note that Optimization I had no observable effect on the value of R_s since the ratio of matches out of total bytes is negligible.

Optimization I uses the *Match Table* data structure that infers space overhead using Snort pattern set of around 2KB per open session. For ModSecurity the space overhead is negligible due to low number of pattern matches within data.

D. Experimental Results Analysis

In this subsection we give intuition for the results described at the above subsections. We calculate a lower bound for the R_s value of the given data set. R_s is described by the following formula,[2] using parameters described in Table II:

$(1 - P_r) + P_r \times ((A_l + A_r)/P_l) + P_r \times M_r \times A_m$

Lower bound of R_s is reached when the following conditions occur: there are no matches at all, where $M_r = A_m = 0$, therefore the third expression becomes zero; minimal value of A_l is zero and the minimal value of A_r is $CDepth$ - 1. Using the optimal $CDepth$ of value 2 we get a ratio of 0.135 which is a *lower bound* for R_s on this data set.

Optimization I skips byte scanning within *Match Segment*s which is represented by A_m. Table II shows that in Snort, Optimization I may improve R_s by at most 4.34% which explains the improvements gained for Snort. In ModSecurity Optimization I do not help due to insignificant M_r.

Another observation from Table II shows that both A_l and A_r are lower at ModSecurity as compared to Snort. That is because Snort has got much more patterns than ModSecurity. Therefore its DFA represents much more prefixes and it is more likely that transitions would lead to depths with higher value. Hence, there are more bytes with $Check$ status and

[2]First expression represents uncompressed bytes which are always scanned. Second expression represents bytes which are scanned from pointers left and right boundaries. Last expression represents bytes which are scanned on *Match Segment*s.

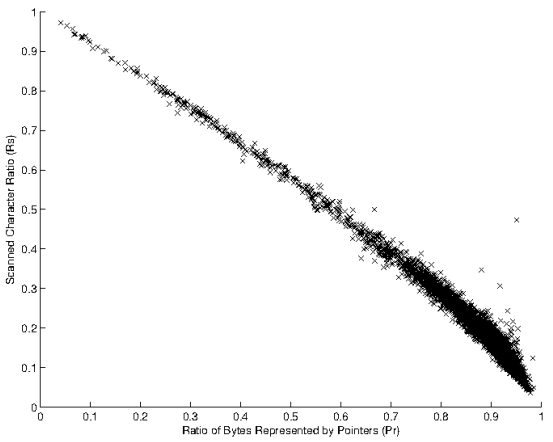


Fig. 7. Relation between the R_s parameter and the P_r as measured for each input HTML page.

more bytes need to be scanned at pointer left and right boundaries.

We note that HTTP traces are usually characterized with high compression ratio. As explained earlier, the skip ratio R_s is strongly related with P_r, the ratio of bytes represented by pointers. Fig. 7 demonstrates the R_s / P_r ratio as measured for each input file separately and shows a linear relationship between those parameters.

E. Using Space Efficient DFA Data Structures

Earlier we claimed that the DFA implementation is orthogonal to ACCH, hence it may be combined with recent state of the art works enjoying their improvements while gaining the additional performance boost achieved by skipping scans with ACCH. In this subsection we support our claim by combining ACCH with some other DFA implementations as stated below.

Up to this point we based the pattern matching engine on the second representation suggested by Aho-Corasick that uses two-dimensional array. This is one of the implementations used by Snort. The first representation proposed by Aho-Corasick indicates explicitly only transitions to the next level of the DFA where for the rest of the possible input bytes it uses a default failure pointer. This representation allows compact implementation of the data structure with the cost of $2n$ transitions for an input of length n.

Recently, a significant effort was made to compact the size of the DFA. Few focused on enhanced hashing schemes such as [31], [32] or another work that suggested BitSplit automata [33]. The above works are aimed at a hardware solution.

In order to experiment ACCH with a more compact DFA implementation we used three different configurations implemented by Snort, namely: Banded, Sparse and Sparse-Banded respectively, for the default failure-transition representation. The ideas of those implementations are mentioned also in recent work by Tuck et al. [15] and Becchi et al. [34]. Table III concludes the normalized throughput of ACCH (with optimizations) as compared to what achieved by the basic, two-dimensional array DFA implementation.

We note that the main focus of recent works is around more efficient *space* representation, mostly in order to allow hardware architecture. Still in software implementations the impact of those works is less significant in the *time* aspect mostly since the space compression is achieved usually by the cost of extra memory references. In addition, Table III shows that the significant performance boost is gained by applying ACCH to either one of the given implementations.

	Memory	**Plain**	**ACCH**
Two-Dimensional Array	74.49MB	1	0.3
Sparse	1.79MB	0.99	0.29
Banded	2.02MB	0.97	0.28
Sparse-Banded	1.89MB	0.98	0.28

TABLE III
NORMALIZED THROUGHPUT OF ACCH (WITH OPTIMIZATIONS) AS COMPARED TO THE TWO-DIMENSIONAL ARRAY DFA IMPLEMENTATION USING SNORT PATTERN-SET.

VIII. CONCLUSION

At the heart of almost every modern security tool is a pattern matching algorithm. HTTP compression becomes very common in today web traffic. Most security tools ignore this traffic and leave security holes or bypass the parameters of the connection therefore harm the performance and bandwidth of both client and server. Our algorithm eliminates up to 84% of data scan based on information stored in the compressed data. Surprisingly, it is faster to do pattern matching on compressed data, with the penalty of decompression, than doing pattern matching on uncompressed traffic. Note that ACCH is not intrusive for the AC algorithm, therefore all the methods that improve AC DFA [12]–[16] 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-pattern matching algorithm on compressed HTTP traffic, and suggest a solution.

ACKNOWLEDGMENT

The authors would like to thank Dr. D. Movshovitz, former VP security Technologies at F5 Networks and I. 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] Port80. [Online]. Available: http://www.port80software.com/surveys/top1000compression
[3] Website optimization, llc. [Online]. Available: http://www.websiteoptimization.com
[4] P. Deutsch, "Gzip file format specification," May 1996, RFC 1952. [Online]. Available: http://www.ietf.org/rfc/rfc1952.txt
[5] ——, "Deflate compressed data format specification," May 1996, RFC 1951. [Online]. Available: http://www.ietf.org/rfc/rfc1951.txt
[6] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, May 1977.
[7] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of IRE*, 1952.
[8] Zlib. [Online]. Available: http://www.zlib.net
[9] A. Aho and M. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 333–340, June 1975.

[10] R. Boyer and J. Moore, "A fast string searching algorithm," *Comm. ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.
[11] (2008, July) Snort. [Online]. Available: http://www.snort.org
[12] T. Song, W. Zhang, D. Wang, and Y. Xue, "A memory efficient multiple pattern matching architecture for network security," in *INFOCOM 2008*, April 2008, pp. 166 – 170.
[13] J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *INFOCOM 2006*. IEEE, April 2006, pp. 1–13.
[14] 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*, July 2007, pp. 186–193.
[15] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memoryefficient string matching algorithms for intrusion detection," in *INFOCOM 2004*, 2004.
[16] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network ids/ips," in *ICNP*, 2006, pp. 187–196.
[17] Y. Afek, A. Bremler-Barr, and Y. Koral, "Efficient processing of multi-connection compressed web traffic," in *NETWORKING 2011, Part I, LNCS*, 2011.
[18] A. Amir, G. Benson, and M. Farach, "Let sleeping files lie: Pattern matching in z-compressed files," in *Proc. of the 5th Symp. on Discrete Algorithms*, January 1994.
[19] 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.
[20] 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.
[21] G. Navarro and J. Tarhio, *Boyer-Moore String Matching over Ziv-Lempel Compressed Text*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, vol. 1848, ch. 16, pp. 166–180.
[22] S. Klein and D. Shapira, "A new compression method for compressed matching," in *DCC 2000*, March 2000, pp. 400–409.
[23] M. Farach and M. Thorup, "String matching in lempel-ziv compressed strings," in *27th annual ACM symposium on the theory of computing*, 1995, pp. 703–712.
[24] L. Gasieniec, M. Karpinski, W. Plandowski, and W. Rytter, "Efficient algorithms for lempel-ziv encoding (extended abstract)," in *In Proc. 4th Scandinavian Workshop on Algorithm Theory*. SpringerVerlag, 1996, pp. 392–403.
[25] U. Manber, "A text compression scheme that allows fast searching directly in the compressed file," *ACM Trans Inf Syst*, vol. 15, no. 2, pp. 124–136, Apr. 1997.
[26] 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*, vol. 42, no. 3, pp. 370–384, mar 2001.
[27] N. Ziviani, E. S. de Moura, G. Navarro, and R. Baeza-Yates, "Compression: A key for next-generation text retrieval systems," *Computer*, vol. 33, no. 11, pp. 37–44, 2000.
[28] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction To Algorithms*. New York: The MIT Press and McGraw-Hill Book Company, 2001.
[29] (2009, July) Alexa. [Online]. Available: http://www.alexa.com
[30] (2008, July) Modsecurity. [Online]. Available: http://www.modsecurity.org
[31] N. Sertac Artan and H. Chao, "Tribica: Trie bitmap content analyzer for high-speed network intrusion detection," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, May 2007, pp. 125 –133.
[32] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, "Hexa: Compact data structures for faster packet processing," in *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*, 2007, pp. 246 –255.
[33] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 112–122. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2005.5
[34] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ser. ANCS '07. New York, NY, USA: ACM, 2007, pp. 145–154. [Online]. Available: http://doi.acm.org/10.1145/1323548.1323573

	Description	Value
P_r	Average ratio of bytes represented by pointers	0.921
P_l	Average pointer size in bytes	16.7
P_h	Average ratio of times that Huffman symbol was longer than 9 bits	0.03

TABLE IV
PARAMETERS ANALYZED USING REAL-LIFE DATA AS DESCRIBED IN SECTION VII.

APPENDIX A
DECOMPRESSION VERSUS PATTERN MATCHING TIME REQUIREMENT

In this section we introduce a simple model that helps us 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.[3]

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., each memory lookup loads the cache with 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 memory.

We start by analyzing the AC algorithm. At the common representation of AC DFA (described at Section II) each state has also a match pointer that points to a matched pattern list in the case it is a "match state" (a.k.a. "output state" [9]) or NULL otherwise. For each byte the AC algorithm performs two memory references, one for next state extraction and the other to check whether a match was found.[4] If the DFA is not in cache memory due to its large size (for example 73MB for 6.6K Snort patterns in 2009), the memory cost for each byte is around 2M. Otherwise, if the DFA fits into the cache, due to a small number of patterns or the usage of DFA compression techniques [12]–[16] 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 common Huffman decoding implementation (as implemented by the "zlib" software library [8]) uses at most two memory lookups per symbol (most of the symbols require only one memory lookup while longer symbols require two memory lookups) as described in Section II. Table IV shows some parameters calculated by using real-life data.

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

[4]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.

	Cache	Memory
AC	$2C$	$2M$
GZIP decompression	$\sim 2C$	$\sim 3C$

TABLE V
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

The number of memory lookups per byte decode at the Huffman stage can be calculated using the following formula[5]:

$P_r \times (P_h + 2)$ / P_l + $(1 - P_r) \times (P_h + 1)$

Using parameters from table IV and assuming Huffman dictionary is in the cache, the overhead of Huffman decoding is $0.19C$ which is negligible.

We now analyze the LZ77 decompression part. For a small number of concurrent sessions where all of the 32KB sliding windows fits into cache, the cost per byte which is not a pointer is $1C$ reference for updating the sliding window. A byte represented by a pointer requires $2C$ for both reading and writing to the sliding window. Since most of the data is represented by pointers we approximate the total overhead caused by LZ77 decoding as $2C$. 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 $P_r * (B_P/P_l * M + 2C) + (1 - P_r) * 2C$ time. Since today the factor between the main memory lookup to cache 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 $3C$.

Table V summarizes the findings. One outcome of this analysis is the observation that when the DFA is in the cache, the GZIP decompression has an equivalent time requirement as the AC algorithm itself. In the case where the DFA is in main memory, the GZIP decompression takes only 7.5% of the time of the AC (assuming $M/C \sim 20$).[6]

APPENDIX B
CORRECTNESS OF ACCH ALGORITHM

In this section we prove Theorem 1 (i.e. that ACCH detects all patterns within given text as AC would), the correctness of ACCH algorithm. The proof relies heavily on the following characteristics of AC DFA: The state of the DFA after reading as input a series of bytes is correlated to the longest prefix of any pattern 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 DFA is equal to k then only the last k bytes of the decompressed traffic are relevant to the state (see Corollary 3).

Formally, let $U_1 \dots U_j$ be the decompressed 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 $P_i = P_{i_1} \dots P_{i_n}$. Let s be the state of AC DFA 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 $U_1 \dots U_j$ is equal to $depth(s)$.

Proof: See [28].

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

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

The following Lemma 4, is the heart of Theorem 1 proof.[7]

Lemma 4 compares between ACCH (Algorithm 2) to the Naïve algorithm (Algorithm 1) that does decompression and AC on all data. Let $Trf = Trf_1 \dots Trf_N$ be the input, the compressed traffic (after Huffman decompression) and let $U_{j_1} \dots U_{j_n}$ be the decompressed 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 DFA.

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 DFA state after scanning the m compressed input (pointer or a byte) is correlated between both algorithms.

Proof: The proof is by induction. For $m = 1$, Trf_1 is the first byte (cannot be a pointer) in the traffic which 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.

For the induction step we assume the claim holds up to $m - 1$, we prove for m.

In the case Trf_m is a byte (and not a pointer), all three claims hold, since after scanning Trf_{m-1} the two algorithms are correlated at the same state (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 go over by induction on the decompressed form of the pointer i.e., on $U_{m_1} \dots U_{m_n}$. For readability let $Pt_1 \dots Pt_{len}$ be equal to $U_{m_1} \dots U_{m_n}$, where $len = m_n$. We will prove the claims on

[5]The left part refers to pointer symbols and the right part refers to literals. Decoding a pointer requires access to two different tables in order to extract both length and distance parameters while decoding a literal requires access to only one table. In case of a symbol which is decoded by more than 9 bits, P_h is added to resemble the overhead of accessing another table.

[6]Experiments on real data with small number of concurrent sessions showed a lower ratio of 3.5%.

[7]Note that we use Lemma 4 for all three versions of $ACCH$. Therefore in each section $ACCH$ refers to the version for which we prove correctness.

each pointer part separately: left boundary, internal and right boundary.

Left Boundary: (Lines 12–18). Based on Claim 3 and the induction assumption, the state of the two algorithms is correlated at pointer start. Both algorithms also receive the same input, hence will have the same state and status for all the input bytes until the Pt_j byte where $Depth(ACCH\_state_{Pt_j}) = Depth(Naive\_state_{Pt_j}) \leq j$. Therefore, all three claims of Lemma 4 hold for the left boundary part. From Corollary 3, we get that from this point the pattern prefix that the $Naive$ and $ACCH$ algorithms try to match is contained within *referred string* (i.e., internal area part).

Internal Area: We prove for two parts, *skipped bytes* (Lines 23–30) and *scanned bytes* (Lines 31–34).

Skipped Bytes: Lines 23–30 skip scanning some bytes, and update the status of those bytes according to the status of the referred bytes. Since we are not at the end of the pointer, we need only to prove the first two claims. *Claim 1:* For every Pt_i ($curPos < i < unchkPos - CDepth + 2$), 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 $Naive\_status_{Pt_i} = Check$, $depth(Naive\_state_{Pt_i}) \geq CDepth$, however the same prefix exists also within referred bytes (see end of left boundary proof). Hence, $Naive\_status_{Pt_{i-dist}} = Check$ (it can be only in state with higher or equal depth). From Claim 1 inductive assumption, $ACCH\_status_{Pt_{i-dist}} = Check$. Since the $ACCH\_status_{Pt_i}$ is copied from the referred bytes (Line 28), i.e., $Check$, we receive contradiction.

Claim 2: For every Pt_i ($curPos < i < unchkPos - CDepth + 2$), iff $Naive\_status_{Pt_i}$ = $Match$ then $ACCH\_status_{Pt_i} = Match$. We first show that there is no $Match$ status in the skipped bytes of $ACCH$, and hence the direction that if $ACCH\_status_{Pt_i} = Match$ then $Naive\_status_{Pt_i} = Match$ is straight forward. From the definition of $unchkPos$ there is no $Match$ in the corresponding bytes at the *referred string*. Since the $ACCH\_status$ in the *pointer* at index $curPos \ldots unchkPos - CDepth + 2$ is the same as the status at the *referred string* (Line 28) 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: (Lines 31–34). The statuses of the $CDepth - 2$ bytes up to index $unchkPos$ are maintained from the *referred string*, 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 $unchkPos + 1$.

Claims 1 and 2: For l where $unchkPos + 1 \leq l \leq matchPos$ we prove that $Naive\_state_{Pt_l}$ and $ACCH\_state_{Pt_l}$ are equal and hence Claims 1 and 2 follow. Since left boundary area scan part is over, all pattern prefixes that have started within *pointer* also exist within the *referred string*, hence have depth equal or lower than the bytes in the *referred string*. Let us look at point $unchkPos$. $unchkPos$ is the maximal index before $matchPos$ where the status of the corresponding byte at the *referred string* is $Uncheck$. It is easy to prove that $Naive\_status_{Pt_{unchkPos}} = Naive\_status_{Pt_{unchkPos-dist}}$ (proof by reductio ad absurdum). From the induction assumption Claim 1, both algorithms statuses are equal at the referred bytes. From definition of $unchkPos$, the status is $Uncheck$ and the depth of both algorithms states is smaller than $CDepth$. Hence, to the AC state relevant only the last $CDepth-2$ bytes (from Corollary 3).[8] Since we reset the DFA state in ACCH (Line 25), we prove in a similar way that only the last $CDepth - 2$ bytes are relevant to the ACCH state. Hence, both algorithms states are correlated at point $unchkPos + 1$. Therefore from this point $Naive\_state_{Pt_l} = ACCH\_state_{Pt_l}$, for any l where $unchkPos + 1 \leq l \leq matchPos$.

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

Now we prove correctness of Theorem 1 based on the correctness of Lemma 4. We show that $ACCH$ detects all patterns in P in the decompressed traffic form of Trf.

Proof: Our proof proceeds by reductio ad absurdum. We assume that the first pattern that $ACCH$ misses is $P_x \in P$ that ends at Trf_m and Pt_i in the decompressed form. From the correctness of AC algorithm, we know that $Naive\_status_{Pt_i} = Match$ which implies that $ACCH\_status_{Pt_i} = Match$ (based on Lemma 4 Claim 2). Therefore at least one pattern P_y was detected by $ACCH$ at Pt_i. P_x and P_y both end at the same position, hence one is a suffix of the other pattern. Since $ACCH$ detected P_y and did not detect P_x we can derive that P_y is a suffix of P_x and that P_x is longer than P_y. Both algorithms states are correlated at pointer start based on Claim 3 on Lemma 4, therefore ACCH could not miss P_x at that part. Thus we get that P_x must be contained within Pt. That means that P_x was contained also at the referred bytes and therefore was detected by $ACCH$. Let s be the DFA state that ACCH reached at the referred bytes after detecting P_x, hence $depth(s) \geq length(P_x)$. This implies that the statuses of the $length(P_x) - CDepth$ bytes prior to Pt_i is $Check$ and that ACCH started the scanned bytes part of the internal area of the pointer, at position $i - length(P_x)$. That implies that ACCH performed DFA transition on all P_x bytes and did not detect it. Thus AC algorithm is not correct and this is a contradiction. ∎

[8] In case where $CDepth$=1, the byte at $unchkPos$ is of depth 0 therefore we don't need to scan any byte, and ACCH is correlated only by the state reset in Line 25.

APPENDIX C
CORRECTNESS OF ACCH OPTIMIZATIONS

A. Correctness of Optimization I

Proof: We prove the correctness of Optimization I of ACCH by showing that Lemma 4 claims still hold for this optimization. A claim that contains ACCH refers to ACCH Optimization I version.

We use induction as in ACCH correctness proof. The induction step, we assume the claim holds until $m-1$, we prove for m. Optimization I changes the way ACCH handles the internal area part. Therefore Lemma 4 claims hold in the cases where Trf_m is a byte and where Trf_m is a pointer it holds for the left and right boundaries parts. For the internal area part we need to prove the correctness of claims 1 and 2.

Claim 1: In Line 10 we update status of the bytes we skip according to the status of the referred bytes (unless the status of the referred byte was $Match$ which is relevant to Claim 2). Therefore proving correctness of Claim 1 for this case is the same as the proof for the *skipped bytes* part for ACCH.

Claim 2: First we show that there cannot be a case where $Naive\_status_{Pt_i} \neq Match$ and $ACCH\_status_{Pt_i}$ = $Match$. If $Naive\_status_{Pt_i} \neq Match$ then there is no pattern contained in Pt that ends at position Pt_i. $ACCH\_status_{Pt_i} = Match$ implies that the status of the referred byte was also $Match$ (since we are at the internal area part). Therefore $Naive\_status$ of that referred byte is also $Match$ from the induction assumption and Claim 2. According to Lines 3–6, if $ACCH\_status = Match$ then the $ACCH\_state$ of the referred byte detects at least one pattern which is contained within referred bytes. That implies that there is a pattern that was fully copied from referred bytes that ends at Pt_i. We receive a contradiction.

Now we show that there cannot be a case where $Naive\_status_{Pt_i} = Match$ and $ACCH\_status_{Pt_i} \neq Match$. If $Naive\_status_{Pt_i} = Match$ then the referred byte status is $Match$ (since we are at the internal area part) and the $Naive\_state$ of the referred bytes detects patterns that are equal or shorter than i. From the induction assumption and Claim 2 we get that $ACCH\_status$ of the referred byte is also $Match$ and it detects all patterns that are detected by $Naive\_state$. Since $ACCH\_state$ at the referred bytes detects at least one pattern that is not longer than i and according to Lines 5 – 6 we get that $ACCH\_status_{Pt_i} = Match$ and therefore receive a contradiction. ∎

B. Correctness of Optimization II

To prove the correctness of Optimization II, we show that Lemma 4 claims still hold. We set the $CDepth$ parameter of $Naive$ algorithm to be equal to $CDepth2$ of Optimization II. We assume that Optimization I is also used in order to keep proof clearness. The enhancement of this proof to a version without Optimization I is straight forward.

Proof: The induction follows the same as in the previous proofs. The delict case is when checking for right boundary, the first byte at the referred bytes with status that is not $Check$ equals $Uncheck1$ (for $Uncheck2$ the algorithm behaves the same as in the previous versions since we set the $CDepth$ parameter to be equal to $CDepth2$). We show correctness of Lemma 4 for this case.

Claim 1: Based on the induction assumption, we know that all the *skipped bytes* statuses which are copied from the referred bytes follow Claim 1. The *scanned bytes* area statuses are set by $scanAC$. Since $scanAC$ sets status $Check$ if $depth \geq CDepth2$, Claim 1 also follows for this area.

Claim 2: This claim follows since from the definition of right boundary there is no $Match$ status there.

Claim 3: Here we show that $Naive\_state_{U_{m_n}} = ACCH\_state_{U_{m_n}}$ (referring to ACCH with Optimization II) also follows where right boundary is determined by $Uncheck1$ status at the referred bytes rather than $Uncheck2$. If we use a version of $Naive$ where the $CDepth$ parameter equals $CDepth1$, Claim 3 correctness is straight forward. Note that the $CDepth$ parameter does affect states of the $Naive$ algorithm since it uses plain AC. Therefore $Naive\_state_{U_{m_n}}$ is the same for either value of $CDepth$. Hence $Naive\_state_{U_{m_n}} = ACCH\_state_{U_{m_n}}$ for $CDepth = CDepth2$, and Claim 3 follows. ∎

Anat Bremler-Barr Anat Bremler-Barr received her B.Sc. degrees in mathematics and computer science (Magna Cum Laude) at 1994, an LL.B degree in law at 1999, and M.Sc (Summa Cum Laude) 1997 and P.hD degree (with distinction) in computer science at 2002, all from Tel Aviv University. In 2001 Anat co-founded Riverhead Networks Inc. in Tel-Aviv, a company that provides systems to protect from Denial of Service attacks. She was the chief scientist of the company. The company was acquired by Cisco Systems in 2004. She joined the school of computer science at the Interdisciplinary Center, Herzliya in 2004. Anat research interests are in computer networks and distributed computing. Her current works are focused on Designing routers and protocols that support efficient and reliable communication.

Yaron Koral Yaron Koral received his B.A. degree in computer science and economics at 1996, a M.B.A. degree in business administration at 2001, all from Tel Aviv University, and M.Sc degree in computer science at 2009 from the Interdisciplinary Center, Herzlia. He is currently a P.hD student working on "Deep-Packet Inspection" networking algorithms, at Tel Aviv University. Yaron research interests are in computer networks and computer security. His current works are focused on multi-pattern matching techniques for intrusion detection systems.