The Interdisciplinary Center, Herzlia

Efi Arazi School of Computer Science

Accelerating Multi-Patterns Matching on Compressed HTTP Traffic

M.Sc. Dissertation for Research Project

Submitted by Yaron Koral

Under Supervision of Dr. Anat Bremler-Barr (IDC)

February 24, 2010
Abstract

'Signature based' detection, based on string-matching algorithms, is a technique used by network security tools to detect malicious activities. Their performance is determined by the speed of the underlying string matching algorithms. Moreover, current security tools do not deal directly with 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 WAF signatures patterns, we show that up to 84% of the data can be skipped in its scan. Surprisingly, we show that it is faster to do pattern matching on the compressed data, with the penalty of decompression, than running pattern matching 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.
List of Figures

2.1 Example of LZ77 compression on Yahoo! home page .................................... 3
2.2 Example of Aho Corrasick DFA ................................................................... 5
3.1 Pointer characteristics distribution .............................................................. 6
5.1 Example of ACCH run for CDepth=2 ............................................................. 12
7.1 Experimental results ..................................................................................... 22
List of Tables

7.1 Parameters of ACCH run using real-life data ........................................... 25
A.1 Parameters of real-life GZIP compressed data ........................................... 28
A.2 Summary of the analysis of time complexity of AC and GZIP decompression .. 29
## List of Algorithms

1. Naive Decompression with Aho-Corasick pattern matching .......................... 8  
2. Aho-Corasick based algorithm on Compressed HTTP ................................. 14  
3. ACCH - Optimization I ........................................................................... 18  
4. ACCH - Optimization II ........................................................................... 19
Chapter 1

Introduction

Security tools, such as Network Intrusion Detection System (NIDS) or Web Application Firewall (WAF) uses signature-based detection techniques to identify malicious activities; namely security tools alert when signatures, a predefined 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 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]. Compressed HTTP is usually used on the server response side since generally server responses contain most of the data.

Multi-patterns matching on compressed traffic is a difficult problem. It requires two time-consuming phases, namely traffic decompression and pattern matching. Therefore, most current security tools either do not scan compressed traffic, which may be the cause of miss-detection of malicious activity or ensure that there is not 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. The last 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 then, in the case of pointers, use this data in order to understand if there is a possibility of finding a match or one can skip scanning this area. We show that ACCH can skip up to 84% of the data and boost the performance of multi-patterns matching algorithm by up to 74%.
Chapter 2

Background

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

LZ77 Compression - The basic idea of the LZ77 compression technique is to compress a series of bytes (characters) 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 [1,32768] 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 (4,6) implies one should return 6 bytes and copy 4 bytes from that point.

Figure 2.1 shows example of Yahoo! home page after compression (Stage I - LZ77). The original page size is 109,143 bytes, while the compressed size after both GZIP stages is 31,716 yielding 73% size reduction. Most of the saving is towards the end of the page, where almost the entire text is encoded by pointers.

Yahoo Decompressed file:

```html
<!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>
</head>
</html>
```

Yahoo LZ77 form:

```html
<!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>
</head>
</html>
```

(a) (b)

**Figure 2.1:** Example of the LZ77 compression on the beginning of the Yahoo! home page (a) Original (b) After the LZ77 compression
Note that the process of 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 provide to facilitate the translation of binary codewords to bytes.

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 bit in a table of $2^9$ entries that represents all possible inputs; each entry holds the symbol value and the actual length used by the symbol. If a symbol exceeds 9 bits, there is an additional reference a second lookup table. Thus, on most of the cases, decoding frequent symbols requires only one memory reference per symbol, while for the less frequent symbols it requires two memory reference.

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

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.

The construction of AC's DFA is done in two phases. First, the algorithm builds a trie of the pattern set: All the patterns are added from the root as chains, where each state corresponds to one symbol. When patterns share a common prefix, they also share the corresponding set of states in the trie. In the second phase additional edges are added to the trie. These edges deal with situations where the input does not follow the current chain in the trie (that is, the next symbol is not an edge of the trie) and therefore we need to transit to a different chain. In such a case, the edge leads to a state corresponding to a prefix of another pattern, which is equal to
the longest suffix of the previously matched symbols.

A common approach in software stores the rules in a two dimensional matrix (see Snort implementation [11]), where a row represents a current state and a column represents a possible symbol. Thus, upon inspecting a symbol of the input, the algorithm reads entry number \((s_i, x)\) and obtains the next state \(s_j\). Figure 2.2(a) depicts the DFA for the patterns set: 'abcd', 'nba', 'nbc'. The resulting automaton has 9 states (corresponding to the nodes of the graph). For readability, we show only 9 transitions (corresponding to the edges of the graph) out of the total \(9 \cdot 256\) transitions. We show only transitions to depth 2 and above, and omit all transitions to depths 1 and 0.

![Figure 2.2: Aho Corasick DFA for patterns: 'abcd', 'nba', 'nbc'. The number inside every state indicates its depth.](image)

Note that this common encoding requires a large matrix of size \(|\Sigma| \cdot |S|\) where \(\Sigma\) 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 suction, require \(16.2MB\) for 1,202 patterns that translate into 16,649 states. There are many compression algorithms for the DFA (for example [12, 13, 14, 15, 16]) but most of them are based on hardware solutions and still have a significant memory requirement.

At the bottom line, DFAs require significant amount of memory, therefore they are usually maintained in main memory and characterize by random rather than consecutive accesses to memory.
Chapter 3

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

This chapter provides an overview of the obstacles in performing multi-patterns matching on compressed HTTP traffic. Note that dealing with on-the-fly 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' on the ongoing web traffic.

We note that there is no apparent "easy" way to perform multi-patterns matching over compressed traffic without decompressing the data in some way. This is mainly because LZ77 is an adaptive compression algorithm; namely, the text represented by each compression 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 since it will not appear in the compressed text in some specific form. For example, the pattern 'abcd' can be expressed in the compressed data by $abc + (j + 3, 3)d$ for all possible $j < 32765$. On the other hand, the following diagrams show the distribution of pointer characteristics for the real-life dataset of Chapter 7:

![Graph](attachment:image.png)

(a) Distance of a pointer (b) Length of a pointer.

Figure 3.1: Distribution of the following pointer characteristics on the real life data set of Chapter 7 (a) distance of a pointer (b) length of a pointer.
hand, Huffman encoding, is non-adaptive within a given text and the same pattern will always be encoded to the same bit string. However, the combination of the two algorithms is therefore adaptive.

The naïve way of performing multi-patterns matching on the traffic, in real time, 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 decompressed 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 straightforward 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). Figure 3.1(a) shows that indeed the distribution of pointers on real-life data set (see Chapter 7 for details on the data set) is spread across the entire window. On the other hand, pattern matching of non-compressed traffic requires only storing one or two packets (to handle cross-packet data), where the maximum size of TCP packet is usually 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, needs 1GB memory while a high-end firewall with 300K concurrent sessions needs 10GB memory. This memory requirement, has implication on not only the price and feasibility of the architecture but also on the capability to perform caching. This work does not handle the challenges imposed by the space aspect and leave it for further research.

**Time** - Recall that pattern matching is already 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 32KB sliding window and therefore has low read-per-byte cost. other hand, the AC algorithm employs a very large DFA that is accessed with random memory reads; this DFA typically does not fit in cache thus requiring main memory accesses.
**Algorithm 1** Naive Decompression with Aho-Corasick pattern matching

$Trf$ - the input, compressed traffic (after Huffman decompression).

$SlideWin_{0...32KB}$ - the sliding window of LZ77, contains last 32KB of decompressed bytes.

$SlideWin_j$ - the decompressed byte which is located $j$ bytes prior to current position.

$DFA(state, byte)$ - AC DFA receives state and byte as input and returns the next state, where $startStateDFA$ 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 $Match(nextState) \neq NULL \text{ then}$
4: \quad act according to $Match(nextState)$
5: end if
6: return $nextState$

7: procedure GZIPDecompressPlusAC($Trf_1 \ldots Trf_n$)
8: $state = \text{startStateDFA}$
9: for $i=1$ to $n$ do
10: \quad if $Trf_i$ is pointer ($dist, len$) \text{ then}
11: \quad \quad for $j = 0$ to $len-1$ do
12: \quad \quad \quad state = scanAC($state, SlideWin_{dist-j}$)
13: \quad \quad end for
14: \quad \quad $SlideWin_{0...len-1} = SlideWin_{dist...dist-len-1}$
15: \quad else
16: \quad \quad state = scanAC($state, Trf_i$)
17: \quad \quad $SlideWin_0 = Trf_i$
18: \quad end if
19: end for
Appendix A introduces a model that compares the time requirements of the decompression and the AC algorithm. Experiments on real data shows 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%.
Chapter 4

Related Work

The problem of pattern matching on compressed data has received attention in the context of the Lempel-Ziv compression family [17, 18, 19, 20]. 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 Chapter 2). Hence all the above works are not applicable to our case. Klein and Shapira [21] 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.

The paper [22] 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 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 in other compression algorithms (not LZ77 which is our case) a similar conclusion was shown in another contexts. For example, it was shown that compressing a file once and then performing pattern matching on the compressed file accelerate the scanning process [23, 24, 25].
Chapter 5

Aho-Corasick based algorithm for Compressed HTTP (ACCH)

In this chapter, 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 we cannot skip the entire pointer area, even if no pattern was matched during the scan of the referred bytes. This is due to the fact that a pattern may occur at a 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. 5.1). 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. 5.1).

Intuitively, detecting a pattern at the left boundary can be done by continuing the scan up to a certain point of the pointer (discussed later). Detecting a pattern at the right boundary or detecting a whole referenced pattern, requires storing information about the previous scans. One natural option is to store the state of each scanned byte at the DFA. However, this is not feasible since we 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 information about the depth of the scanned byte state at the DFA. 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 DFA. 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 is contained in the pointer area by comparing the matched state
Figure 5.1: Example of ACCH run for $CDepth=2$. The example is based on the DFA from Fig. 2.2. 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. The blue rectangles indicate the regions of the left and right boundaries of the pointer.

depth to the matched byte location at the referred bytes. If the depth is equal or smaller than the location at the referred bytes area, then we can be certain that the pattern was fully copied by the pointer and therefore there is a match within that location in the pointer area. Once again, storing the exact depth information is not feasible, since we cannot calculate the exact 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 DFA 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 DFA. The two other statuses of a byte are estimated indicators of the depth of the scanned byte state at the DFA in the case where the status is not Match. As recalled, the estimation is required, since we don’t have 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. Our status would indicate only if the depth is above or below $CDepth$. The $CDepth$ parameter represents "low depth". Its optimal value is determined using the experiments described on Chapter 7. 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 we are certain that the depth of the state is below $CDepth$. Every byte with Uncheck status raises the chance for skipping more bytes within pointer area. As noted in [12, 13], most of the time the DFA 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 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 case of skipped bytes.
Still, mistake in that direction does not cause misdetection. It only causes unnecessary Check statuses which means we skip less bytes.

We now give the details of the algorithm (see detailed pseudo code in Algorithm 2). We define scanAC as a the main function of AC. It receives as an input the 'current state' (of the DFA) and a byte and return the ‘next state’ and ‘byte status’ by performing AC DFA transitions (see lines 1-11). In ACCH we skip calling scanAC for some of the bytes.

ACCH handles 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 Figure 5.1).

**Left Boundary:** In order to detect a pattern that is in the left boundary of pointer area (see function scanLeft, lines 12-18), it is enough to continue scanning with scanAC until we reach a byte in the pointer area, where the depth of the state we reach in the DFA after scanning the that byte, is less or equal to the number of bytes in the pointer area we already scanned. There is no need to continue the scan, since from that point matches within pointer area indicate a pattern which is fully contained within pointer area which is dealt with in the next case.

**Internal area:** In order to detect patterns that are fully contained within pointer area (lines 41-47), we need to check if there is a byte with Match status in the referred bytes. Note that a Match in the referred bytes indicates that there is a possibility of a pattern contained 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 referred by pointer area.

We would like to scan as few bytes as possible in order to detect whether the pattern actually occurs at that position in the data. Note that in order to detect a pattern of length k, AC needs to scan only k bytes. We use the Check/Uncheck status to determine how many bytes before the byte with the Match status we need to scan. We call the area between current position up to the first Match status a MatchSegment. Let matchPos be the first index of the byte that has a Match status within internal area. Let unchkPos be the the maximal index of the byte that has Uncheck status and unchkPos < matchPos. It is easy to see that if there is a pattern within this area, it can start not before the unchkPos – CDepth + 2 position in the referred bytes, since otherwise we would have a contradiction to the definition of unchkPos. Hence we skip scanning bytes from the beginning of the MatchSegment (last byte scanned) up to the position of unchkPos – CDepth + 2 (see function scanSegment lines 20-35). This is the saving by our algorithm and the way we achieve the performance improvement. The challenging part is to maintain 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 referred bytes in the sliding window (see line 28). If a status of a byte in the internal 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 DFA would have been (if
**Algorithm 2** Aho-Corasick based algorithm on Compressed HTTP

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 \( \text{byteInfo} \).
- \( \text{SlideWin}_j \) - Represents the \( j^{th} \) byte prior to current position in the sliding window.
- **Depth(state)** - a function that returns the depth of the state in the DFA.
- \( CDepth \) - the constant parameter of ACCH algorithm.

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

12: \( (\text{nextPos}, nextState) = \text{function scanLeft}(state, \text{curPtrInfo}) \)
13: \( \text{curPos} = 0 \)
14: while \( (\text{Depth}(state) > \text{curPos}) \land (\text{curPos} < \text{len}) \) do
15: \( (status, state) = \text{scanAC}(state, \text{SlideWin}_{\text{dist-curPos}}.b) \)
16: \( \text{curPtrInfo}[\text{curPos}].status = status \)
17: \( \text{curPos}++ \)
18: end while
19: return \( (\text{curPos}, state) \)
function scanSegment(state, start, end, curPtrInfo)

Find the maximal unchkPos, start ≤ unchkPos ≤ end such
SlideWin_{dist−unchkPos}.status = Uncheck

if start < (unchkPos − CDepth + 2) then
    Skip and update window
else
    curPtrInfo[start...(unchkPos − CDepth + 2)].status = SlideWin_{dist−start...(dist−(unchkPos−CDepth+2))}.status

state = startStateDFA
for curPos = (unchkPos − CDepth + 2) to (unchkPos) do
    Scan bytes and copy status from SlideWin
(state, status) = scanAC(state, SlideWin_{dist−curPos}.b)
curPtrInfo[curPos].status = status
end for
for curPos = unchkPos + 1 to end do
    Internal or Right Boundary scan
(state, status) = scanAC(state, SlideWin_{dist−curPos}.b)
curPtrInfo[curPos].status = status
end for
return (curPos+1, state)

procedure ACCH(Trf1...Trf_n)
for i=1 to n do
    if Trf_i is pointer (dist,len) then
        curPtrInfo[0...len−1].b = SlideWin_{dist...dist−len}.b
    (state, curPos) = scanLeft(state,curPtrInfo)
    while curPos < len do
        Check Matches within internal area
        Find the minimal matchPos, curPos ≤ matchPos < len
        such SlideWin_{dist−matchPos}.status = Match
        if no such matchPos exist then
            Case of Right Boundary Segment
            curPos=scanSegment(state, curPos, len-1, curPtrInfo)
        else
            Case of Match Segment
            curPos=scanSegment(state, curPos, matchPos, curPtrInfo)
        end if
    end while
    slideWin_{len−1...0} = curPtrInfo[0...len−1]
    else
        Trf_i is not a pointer, performing simple byte scan
        SlideWin_{0}.b = Trf_i
    (state, status) = scanAC(state, SlideWin_{dist−i}.b)
    SlideWin_{0}.status = status
end if
end for
we run the naive algorithm, i.e. run scanAC on all the bytes), is correlated to a prefix that is internal to the pointer area and hence was copied completely from referred bytes, therefore was also detected there. 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 suffix of a pattern that started prior to the referred bytes area. Hence, in case of a pointer to that area, we might perform some scanAC redundant calls because of bytes that were marked with status Check where their true status should have been Uncheck. However these redundant calls do not harm the correctness of the algorithm.

In order to determine whether there is a match, we try to correlate the state of the DFA with the one of the naive algorithm (i.e. Algorithm 1) at that point. For that, we set DFA to start state and continue scanning from unchkPos − CDepth + 2. Note that statuses for the area between unchkPos − CDepth + 2 to unchkPos were maintained from related bytes (lines 26-29). As a rule scanAC gives more accurate information about the real status of the bytes, however, at these CDepth bytes before unchkPos the status of scanAC may be misleading - since we start from the start state of DFA, it will always return Uncheck status even though it may not be the case. After the first CDepth bytes we continue scanning up to matchPos, this time we use the statuses that returned from scanAC (lines 31-34).

We then repeat handling MatchSegments until there are no more matches within internal area (line 41). Then we check if there is a pattern at the pointer right boundary in a similar way to the previous case by calling scanSegment again.

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. Therefore, 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 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 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 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.
Chapter 6

ACCH Optimizations

In this chapter we present two techniques that exploit the characteristics of ACCH further to achieve even better performance than ACCH as presented by Algorithm 2. ACCH algorithm rescans \( CDepth - 1 \) bytes at every right boundary or a Match byte. This implies that when the data contains many “matches”, ACCH performance reduces dramatically. Optimization I fits for usage in data with many ”matches”. It eliminates the need for rescanning bytes for matched patterns. Optimization II refers to another aspect of ACCH. Note that the algorithm uses only three statuses out of the four that it could, using 2 bits. Optimization II uses a fourth status to achieve better performance.

6.1 Optimization I - removing effect of matches within pointer area

In algorithm 2, scans occurred on pointer left boundary, right boundary and on the internal area at MatchSegment. Each MatchSegment resulted with expensive accesses to DFA. The idea behind Optimization I, is to avoid the internal scans required for MatchSegments and scan only pointer left and right boundaries.

In order to do that, we use a hash table called MatchTable. Whenever scanAC reaches a ”match state”, the algorithm stores all patterns that were found by this state in MatchTable and use the absolute position of the byte in the data as a key to that entry in the hash table.

The general outline of Algorithm 3 is built from three stages: ”scan left boundary”, ”handle internal matches” (see lines 1-12) and ”scan right boundary”. The second stage is performed by checking for each of the Match statuses found within pointer area, the list of its related patterns in MatchTable. For each pattern in that list with length shorter or equal to the index of the ”match status”, we can determine that it was copied completely and therefore is contained within pointer area. If there is at least one pattern which is contained within pointer area, we mark the status as Match and add all patterns that are not longer than the index of the current Match (see lines 5-6).

A case where all patterns are longer than the index of the current Match implies that only
Algorithm 3 ACCH - Optimization I

absPosition - Absolute position from the beginning of data.

After line 38: \( \text{absPosition} += \text{len} \)

After line 49: \( \text{absPosition} ++ \)

MatchTable - A hash table, where each entry represents a Match. The key is the Match \( \text{absPosition} \) and the value is a list of patterns that were located at that position.

Function \text{scanAC} - a new line is added after line 4:
add patterns in \( \text{Match(state)} \) to \( \text{MatchTable(\text{absPosition})} \)

Procedure ACCH - instead of the while loop, lines (41-47):
\( \text{handleInternalMatches(state, curPos, len-1)} \)
\( \text{scanSegment(state, curPos, len-1)} \)

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

1: function handleInternalMatches(start, end)
2: for curPos = start to (end) do
3: \quad if SlideWin_{\text{dist}}-\text{curPos}.\text{status} = \text{Match} then
4: \quad \quad if MatchTable(curPos) contains patterns
5: \quad \quad \quad shorter or equal to curPos then
6: \quad \quad \quad \quad add those patterns to MatchTable(absPosition)
7: \quad \quad curPtrInfo[curPos].\text{status} = \text{Match}
8: \quad \quad else curPtrInfo[curPos].\text{status} = \text{Check}
9: \quad \quad end if
10: \quad else
11: \quad \quad curPtrInfo[curPos].\text{status} = SlideWin_{\text{dist}}-\text{curPos}.\text{status}
12: \quad end if
13: end for
the suffix of those patterns was copied therefore there is no match in that position within pointer area. Since we are at the internal area part and since no pattern that ends at that position was copied completely, the status of that position is not \textit{Match} and cannot be maintained from the referred bytes. Therefore we set the status to \textit{Check} (line 7) since an error in that direction does not harm algorithm correctness (as discussed in Chapter 5). All statuses of skipped bytes within internal area, other than matches, are copied from the referred bytes (line 10).

This optimization gains a significant speed improvement as shown in the experimental results chapter. The downside of this optimization is the additional data structure \textit{MatchTable} that has to be maintained.

### 6.2 Optimization II - using additional status

ACCH, as described in algorithm 2, uses three possible statuses for each byte: \textit{Check}, \textit{Uncheck} and \textit{Match}. Those statuses are maintained within the sliding window using two bits. Since two bits can represent four different values, there can be one more status 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. The additional status is ”for free” with no need for extra memory because of the fact that two bits can represent four statuses and we used only three. This is done by defining two constant parameters: \textit{CDepth1} and \textit{CDepth2} instead of one, where \textit{CDepth1} < \textit{CDepth2}. In the case of a byte without ”match state”, the status is determined the following way: if depth is smaller than \textit{CDepth1} then the status is \textit{Uncheck1}. If it is smaller than \textit{CDepth2} but not from \textit{CDepth1} then the status is \textit{Uncheck2}. Otherwise the status is \textit{Check}.

**Algorithm 4 ACCH - Optimization II**

\textit{CDepth1}, \textit{CDepth2} - Instead of one constant parameter \textit{CDepth}, we maintain two, where \textit{CDepth1} < \textit{CDepth2}

Function \textit{scanAC} - line 8 changes to lines:

\begin{verbatim}
else if Depth(state) \leq CDepth1 then status = Uncheck1
else status = Uncheck2
\end{verbatim}

Function \textit{scanSegment} - line 21: instead of searching for maximal \textit{Uncheck}, it searches for maximal \textit{Uncheck1} or \textit{Uncheck2}

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

The tradeoff of using \textit{CDepth}, as discussed before, is between using a higher value of \textit{CDepth}
and gain more Uncheck statuses (which in turn leads to more skipped bytes) or lower value of CDepth and scan less bytes at every scanSegment call prior to unchkPos. By using two constant depth estimators instead of one, the algorithm gains the advantages of both cases (see results at the experimental results presented at Chapter 7). The algorithm mark as much Uncheck1 and Uncheck2 statuses as it would mark Uncheck statuses in the case where CDepth from algorithm 2 equals CDepth2. And scanSegment scans the same amount of bytes it would have if CDepth from algorithm 2 would have been equal to CDepth1. Status updates are maintained at the same logic as in algorithm 2, either copying statuses from referred bytes or determining the status with scanAC according to state depth.
Chapter 7

Experimental Results

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

7.1 Data Set

In order to evaluate ACCH performance we need two data sets, one of the traffic and the other of the patterns. We used a list of the most popular web pages taken from Alexa web site [26], that maintains web traffic metrics and top sites lists. Over 37K URLs were browsed and only those that were encoded by GZIP were stored for our traffic data set. The data set contains 14,078 compressed HTML pages that takes 808MB in decompressed form and 160MB in the compressed 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 16.7.

As a data set of signatures, we use two sets: one of the ModSecurity [27], an open source web application firewall, and one of Snort, an open source network intrusion prevention and detection system [11].

In ModSecurity we choose signatures group that applies to HTTP-response (only the response is compressed). Patterns are normalized in such a way that they will not contain regular expressions.\(^\text{1}\) Total number of patterns is 124.

The Snort data set, taken from the published rules on June 2008 [11], 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 HTTP 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

\(^1\text{Regular expressions were extracted into several plain patterns.}\)
Figure 7.1: (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 naive 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.
in traffic data, since Snort has patterns like "st", "id=" or "ver". The data set contains 655 occurrences of ModSecurity patterns and over 14M of Snort patterns.

7.2 ACCH performance

In this section, 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 7.1(a) summarizes $R_s$ as a function of $CDepth$, for Snort and ModSecurity patterns. As shown in Figure 7.1, $CDepth$ 2 shows best performance for both pattern sets. $R_s$ for Snort patterns is 0.27 and for ModSecurity is 0.181. 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 pointer right boundary. Snort patterns contains a significant amount of matches (near 2% of the total number of bytes are with status Match). 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 234,448 matches (instead of 14M). As can be seen from 7.1(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 a much more significant influence than number of patterns on $R_s$ value.

Figure 7.1(b) shows the correlation between the $R_s$ and the performance benefit. We use Intel Core 2 Duo 2GHZ with 2Giga RAM as a platform. The "Plain" column shows the performance ratio of running AC on plain uncompressed data compared to the naive algorithm (that requires decompression). ACCH with $CDepth$ of value 0 marks all statuses as Check, therefore no byte scan skips takes place. There is a slight overhead in implementing the algorithm compared to plain AC, which is between 4%-8%. It can be explained by the common task of both naive algorithm and ACCH of decompressing the data (around 4% of the overhead) and the additional memory used for the statuses of the bytes in the sliding window. The statuses cause a larger GZIP data structure which in turn results in more memory references. For $CDepth = 2$, ACCH running over ModSecurity achieved 74% performance improvement and ACCH running over Snort achieved 64% performance improvement.

It is easy to see from the algorithm that two factors influence the scanned character ratio, the ratio of matches (Optimization I removes the influence of this factor), which is usually low for security tools, and the ratio of "Uncheck" statuses, the number of bytes in the traffic that after DFA scan reach states close to DFA root. As noted before and analyzed by [12, 13] most of the time the DFA uses states close to DFA root.

The surprising finding is that we work faster on the compressed files than on uncompressed ones, as can be shown by comparing the "Plain" column in figure 7.1(b) to other columns with $CDepth$ higher than 0.
7.3 ACCH Optimizations

In order to analyze the benefit of ACCH optimizations as presented in Chapter 6, we used the Snort pattern set. Recall that the Snort data set resulted in around 14M Match statuses, therefore it is a good candidate to measure the effectiveness of the proposed Optimization I. For Optimization II we used all combinations of $CDepth_1$ and $CDepth_2$ values. It turned out that the optimal value was where $CDepth_2 = CDepth_1 + 1$. In figure 7.1(c) shows four combinations where $CDepth_2 = CDepth_1 + 1$. The values of the x-axis refer to the $CDepth_2$ value, where in the first column $CDepth_1 = CDepth_2 = 1$. Figure 7.1(c) shows the impact on $R_s$ for both optimizations compared to the original ACCH algorithm. For Optimization I, $CDepth$ of value 2 is optimal with $R_s$ of value 0.229 and for Optimization II $CDepth$ of value 3 is optimal with $R_s$ of value 0.215 as compared to $R_s$ of 0.27 for the original ACCH. When we used the ModSecurity pattern set, $CDepth$ of value 2 was optimal Optimization I and for Optimization II the optimal parameters were $CDepth_1 = 1$ and $CDepth_2 = 2$. For Optimization I we got $R_s$ of 0.181 and for Optimization II we got $R_s$ of 0.163. Note that Optimization I had no observable effect on the value of $R_s$. This is due to the fact that the ratio of matches out of total bytes is negligible therefore removing scans caused by matches (as proposed by Optimization I) has no observable effect on $R_s$.

Optimization I space overhead for Snort pattern set was around 2KB per open session. For ModSecurity the space overhead was negligible due to low number of pattern matches within data.

7.4 Experimental Results Analysis

In this section we give intuition for the results described at the above sections. We try to calculate a lower bound for the $R_s$ value of the given data set for any pattern set. We do that for $CDepth$ of value 2, which was the optimal value for all pattern sets in the experiments. We define few more variables as described in Table 7.1.

$R_s$ can be bounded by the following formula:\footnote{First expression represents decompressed 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 prior to match statuses.}

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

In the case of no matches at all, the $M_r$ and $A_m$ parameters equal zero, therefore the third expression becomes zero. The 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 skip scanning bytes prior to Match statuses, represented by $A_m$. Table 7.1 shows that in Snort, Optimization I may improve $R_s$ by at most 4.34%. This explains the results
<table>
<thead>
<tr>
<th>Param.</th>
<th>Description</th>
<th>Snort</th>
<th>ModSecurity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_l$</td>
<td>Average number of bytes scanned on the left boundary of pointers</td>
<td>1.077</td>
<td>0.684</td>
</tr>
<tr>
<td>$A_r$</td>
<td>Average number of bytes scanned on the right boundary of pointers</td>
<td>1.61</td>
<td>1.159</td>
</tr>
<tr>
<td>$A_m$</td>
<td>Average number of bytes scanned prior to match status</td>
<td>2.55</td>
<td>4.824</td>
</tr>
<tr>
<td>$M_r$</td>
<td>Ratio of bytes marked with $Match$ status by ACCH out of all bytes</td>
<td>0.017</td>
<td>$7.73 \times 10^{-7}$</td>
</tr>
</tbody>
</table>

Table 7.1: Parameters analyzed using real-life data for Snort and ModSecurity. Parameters measured while executing ACCH with $CDepth$ of value 2.

obtained from optimization I for the Snort pattern set. In ModSecurity Optimization I may not help at all due to insignificant ratio of matches out of total bytes.

Another observation from Table 7.1 shows that both $A_l$ and $A_r$ are lower at ModSecurity compared to Snort. This can be explained since Snort has got much more patterns than Mod-Security. Therefore its DFA is more dense at lower depths and it is more likely that DFA scan lead to deeper depths. That means that there are more $Check$ statuses and more bytes need to be scanned at pointer left and right boundaries.
Chapter 8

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, achieves elimination of up to 84% of data scans 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 regular traffic. Note that ACCH is not intrusive for the AC algorithm, therefore all the methods that improve AC DFA [12, 13, 14, 15, 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-patterns matching algorithms on compressed HTTP traffic, and suggest a solution.
Appendix A

Decompression versus Pattern Matching Time Requirement

In this chapter 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 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., 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. At the common representation of AC DFA (described at Chapter 2) 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" \([9]\)) 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 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 \([12, 13, 14, 15, 16]\) 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

---

\(^1\)Using hardware solutions or by special assembly commands that give recommendation to the loader.

\(^2\)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.
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). See Chapter 2 for Huffman decoding implementation details. Table A.1 shows some parameters calculated by using real-life data.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_r$</td>
<td>Average ratio of bytes represented by pointers</td>
<td>0.921</td>
</tr>
<tr>
<td>$P_l$</td>
<td>Average pointer size in bytes</td>
<td>16.7</td>
</tr>
<tr>
<td>$P_h$</td>
<td>Average ratio of times that Huffman symbol was longer than 9 bits</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table A.1: Parameters analyzed using real-life data as described in Chapter 7.

Number of memory lookups per byte decode at the Huffman stage can be calculated using the following formula:\(^3\)

$$P_r \times \frac{(P_h + 2)}{P_l + (1 - P_r) \times (P_h + 1)}$$

Using parameters from table A.1 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 fit into cache, the cost per byte which is not a pointer is $1C$ reference for updating the 32KB sliding window. A byte represented by a pointer requires $2C$ for both reading and writing to the 32KB 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 $B_P = \lceil (P_l/B) \rceil + (P_l \text{mod} B) / B$. In order to understand the cost per byte we need to divide the result by the average pointer length. Hence we need total $P_r \times (B_P/P_l \times M + 2C) + (1 - P_r) \times 2C$ time. Since today the factor between the regular memory lookup to cache memory lookup of $M/C$ is usually between $10 - 20$ and $B$ is $32\text{Bytes}$, the $P_r \times B_P/P_l$ 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

\(^3\)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.
dominated by the pointer analysis part, hence it is around $3C$.

Table A.2 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 regular memory, the GZIP decompression takes only 7.5% of the time of the $AC$ (assuming $M/C \sim 20$).\(^4\)

<table>
<thead>
<tr>
<th></th>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>$2C$</td>
<td>$2M$</td>
</tr>
<tr>
<td>GZIP compression</td>
<td>$\sim 2C$</td>
<td>$\sim 3C$</td>
</tr>
</tbody>
</table>

**Table A.2:** 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.

\(^4\)Experiments on real data with small number of concurrent sessions showed a lower ratio of 3.5%.
Appendix B

Correctness of ACCH algorithm

In this chapter 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 AC 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 AC 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 $U_1 \ldots U_j$ 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 $P_i = P_{i_1} \ldots P_{i_n}$. 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 $U_1 \ldots U_j$ is equal to $\text{depth}(s)$.

Proof: See [28].

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

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

The following Lemma 4, is the heart of Theorem 1 proof.\(^1\)

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

\(^1\)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.
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. $Naive\_state_{U_{mn}} = ACCH\_state_{U_{mn}}$ i.e., the state after decompressing the $m$ compressed input (pointer or a byte) is correlated.

Proof: The proof is by induction. For $m = 1$, $Trf_1$ is the first byte (cannot be a pointer) in the traffic and decompresses to $U_{11}$. 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 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 $U_{m_1}...U_{m_n}$.

**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 12-18). Based on Claim 3 and on 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 bytes (i.e., internal area part).

**Internal Area:** The proof here is divided to two parts, skipped bytes (lines 23-30) and scanned bytes (lines 31-34).

**Skipped Bytes:** In lines 23-30 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 $Pt_i$ ($cur\_Pos < i < unchkPos − CDepth + 2$), 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_{t-dist}} = \text{Check}. Since the ACCH\_status_{Pt_{t}} is copied from the referred bytes (line 28), i.e., Check, we receive contradiction.

Claim 2: Here we need to prove that for every Pt_{i} \ (\text{curPos} < i < unchkPos - CDepth + 2), \iff Naive\_status_{Pt_{i}} = \text{Match} \ then \ ACCH\_status_{Pt_{i}} = \text{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}} = \text{Match} then Naive\_status_{Pt_{i}} = \text{Match}, is proven straight forward from this fact. From the definition of unchkPos there is no Match in the corresponding referred bytes. Since the ACCH\_status in the referred bytes in index \text{curPos} \ldots unchkPos – CDepth + 2 is the same as the status in the sliding window (line 28) there is no Match in ACCH\_status_{Pt_{i}}. The direction that if Naive\_status_{Pt_{i}} = \text{Match} then ACCH\_status_{Pt_{i}} = \text{Match} is proven in a similar way to the proof of Claim 1.

Scanned Bytes: Here we prove the claims for the bytes in the scanned area (i.e., lines 31-34). The statuses of the \text{CDepth} – 2 bytes up to index unchkPos 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 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 we are after the left boundary area, we are ensured that all pattern prefixes are within pointer boundaries and were copied completely from the referred bytes and hence have depth equal or lower than in the referred bytes. Let us look at point unchkPos. unchkPos is the maximal index before matchPos where the status of the corresponding referred byte at ACCH is Uncheck. It is easy to prove that Naive\_status_{Pt_{\text{unchkPos}}} = Naive\_status_{Pt_{\text{unchkPos}}-dist} \ (\text{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 \text{CDepth}. Hence to the AC state relevant only the last \text{CDepth} – 2 bytes (from Corollary 3).\textsuperscript{2} Since we reset the DFA state in ACCH (see algorithm line 25), we can prove in a similar way that only the last \text{CDepth} – 2 bytes are relevant to the AC 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 \text{end} = matchPos (not including matchPos) there is no referred byte with Match status. Hence the claim also follows for the case where segment \text{end} = len – 1. In this case we need also to prove Claim 3, which we prove 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.

\textsuperscript{2}In case where \text{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.
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.
Appendix C

Correctness of ACCH Optimizations

C.1 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 \textit{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 \textit{Naive} status\( P_i \neq \textit{Match} \) and \textit{ACCH} status\( P_i = \textit{Match} \). If \textit{Naive} status\( P_i \neq \textit{Match} \) then there is no pattern contained in \( Pt \) that ends at position \( P_i \). \textit{ACCH} status\( P_i = \textit{Match} \) implies that the status of the referred byte was also \textit{Match} (since we are at the internal area part). Therefore \textit{Naive} status of that referred byte is also \textit{Match} from the induction assumption and Claim 2. According to lines 3 - 6, if \textit{ACCH} status = \textit{Match} then the \textit{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 \( P_i \). We receive a contradiction.

We show that the case where \textit{Naive} status\( P_i = \textit{Match} \) and \textit{ACCH} status\( P_i \neq \textit{Match} \) cannot be. If \textit{Naive} status\( P_i = \textit{Match} \) then the referred byte status is \textit{Match} (since we are at the internal area part) and the \textit{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 \textit{ACCH status} of the referred byte is also \textit{Match} and it detects all patterns that are detected by \textit{Naive state}. Since \textit{ACCH state} at the referred bytes detects at least one pattern that is not longer than
and according to lines 5 - 6 we get that $ACCH_{status_{Pl_i}} = Match$ and therefore receive a contradiction.

C.2 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 straightforward.

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 $Naive_{state_{U_{mn}}} = ACCH_{state_{U_{mn}}}$ (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 straightforward. Note that the $CDepth$ parameter does affect states of the $Naive$ algorithm since it uses plain AC. Therefore $Naive_{state_{U_{mn}}}$ is the same for either value of $CDepth$. Hence $Naive_{state_{U_{mn}}} = ACCH_{state_{U_{mn}}}$ for $CDepth = CDepth2$, and Claim 3 follows.
Bibliography


תקציר

"więtפת מחרוזות" מתעבורה מ潟וח כוות אשר את בתוכן התוכנית הבסיסית
לאחר פעלויות זרויות על דוי דלי בבוש. התוכנית מ潟וח מירב דומייני
בקביעה בזגועים של כל האבטחה בלשל מרכבות של האגטטמן טhighest
מאותות. זך עם זא, כוות כלא אבטחה אינפ מומדימו עס תעבורה והוסה.charCodeAt
הпуска均价י באטייטט בפיים האקריםות.

עבורה ואומיכות בעבורה HTTP כוות. פרוטוקול HTTP
בচית הדפים חון שימוש-ב, GZIP, אשר מעריך שלפ מקודים של פיתוח הדיתשם
לפיני ביתוי שלב חיתון המ[trainers

HTTP יבוש עבורה Aho Corasick
בעאמר זה אנא מعزيים אתוריים מבוסס בד
דוחות砷 מתגי את המידיע שنجح ב regiment הדפס כוות כי להאיאית את
אלגוריתמים חיתון המראות. בידית העבורה HTTP אמיתית ושימוש בחיתמה
tמשמשות כל אבטחה בשוק, מראה כי האגלוריטם חיתס שירק כי השודק
עד 84% מהתעבורה בתק מקר את ביתוי שלב חיתון המראות באותת המשמעויות.
באמפים מפילים, אנא מretry כי מחיי יהיר ילבצע חיתון המראות על מידי
דוחות עם התוקד המנהג מתגי הדיתשם, מאשר לבלצע את חיתון המראות על
מידי עצאני הים. למלטיז דייטעוז עם העבורה האגטטמן המנהג את הביעה
של ביתוי חיתון המראות על HTTP דוחות בק込む אいたら ומצייע פתרון.
המערכת היבנותמות בהרצילה
בית-ספר אפי ארצי למדעי המחשב

האצת חיפוש מחזורות
מעל גבי תעבורת
דחוסה HTTP

מונח חסיבר סופי לפוריקט מחקר
M.Sc. לשראות תואר MUSIC

על-ידי יורי קורל

העבירה בוצעה בחנויות ד"ר ענת ברמל-בר

פברואר 2010