Shift-based Pattern Matching for Compressed Web Traffic

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

Yaron Koral
Blavatnik School of Computer Sciences
Tel-Aviv University, Israel
Email: yaronkor@post.tau.ac.il

Victor Zigdon
Computer Science Dept.
Interdisciplinary Center, Herzliya, Israel
Email: victor.zigdon@gmail.com

***Abstract*—Compressing web traffic using standard GZIP is becoming both popular and challenging due to the huge increase in wireless web devices, where bandwidth is limited. Security and other content based networking devices are required to decompress the traffic of tens of thousands concurrent connections in order to inspect the content for different signatures. The overhead imposed by the decompression inhibits most devices from handling compressed traffic, which in turn either limits traffic compression or introduces security holes and other dysfunctionalities.**

The ACCH algorithm [1] was the first to present a unified approach to pattern matching and decompression, by taking advantage of information gathered in the decompression phase to accelerate the pattern matching. ACCH accelerated the DFA-based Aho-Corasick multi-pattern matching algorithm. In this paper, we present a novel algorithm, SPC (Shift-based Pattern matching for Compressed traffic) that accelerates the commonly used Wu-Manber pattern matching algorithm. SPC is simpler and has higher throughput and lower storage overhead than ACCH. Analysis of real web traffic and real security devices signatures shows that we can skip scanning up to 87.5% of the data and gain performance boost of more than 51% as compared to ACCH. Moreover, the additional storage requirement of the technique requires only 4KB additional information per connection as compared to 8KB of ACCH.

I. Introduction

Compressing HTTP text when transferring pages over the web is in sharp increase motivated mostly by the expansion in web surfing over mobile cellular devices such as smartphones. Sites like Yahoo!, Google, MSN, YouTube, Facebook and others use HTTP compression to enhance the speed of their content downloads. Moreover, iPhone API for apps development applied support for web traffic compression. Fig. 1 shows statistics of top sites using HTTP Compression: two-thirds of the top 1000 most popular sites use HTTP compression. The standard compression method used by HTTP 1.1 is GZIP.

This sharp increase in HTTP compression presents new challenges to networking devices that inspect the traffic contents for security hazards and balancing decisions. Those devices reside between the server and the client and perform *Deep Packet Inspection* (DPI). When receiving compressed traffic the networking device needs first to decompress the message in order to inspect its payload. This process suffers from performance penalties of both *time* and *space*.

Those penalties lead most security tools vendors to either ignore compressed traffic, which may lead to miss-detection

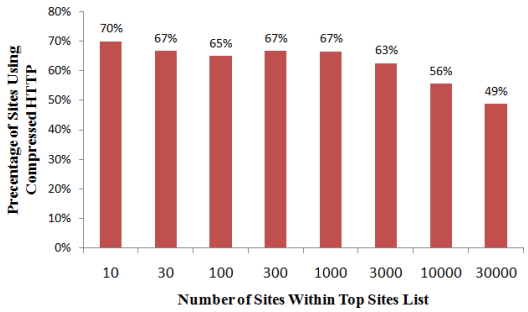


Fig. 1. HTTP Compression usage among the Alexa [2] top-site lists

of malicious activity, or ensure that no compression takes place by re-writing the 'client-to' HTTP-header to indicate that compression is not supported by the client's browser thus decreasing the overall performance and bandwidth. Few security tools [3] handle HTTP compressed traffic by decompressing the entire page on the proxy and performing signature scan on it 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.

Recent work [1] presents technique for pattern matching on compressed traffic that decompresses the traffic and then uses data from the decompression phase to accelerate the process. Specifically, GZIP compression algorithm eliminates repetitions of strings using back-references (pointers). The key insight is to store information produced by the pattern matching algorithm for scanned decompressed traffic, and in case of pointers, to use this data to either find a match or to skip scanning that area. That work analyzed the case of using the well known Aho-Corasick (AC) [4] algorithm as a multi-pattern matching technique. AC has a good worst-case performance since every character requires traversing exactly one Deterministic Finite Automaton (DFA) edge. However, the adaptation for compressed traffic, where some characters represented by pointers can be skipped, is complicated since AC requires inspection of every byte.

Inspired by the insights of that work, we investigate the case of performing DPI over compressed web traffic using the shift-based multi-pattern matching technique of the modified Wu-Manber (MWM) algorithm [5]. MWM inherently does not scan every position within the traffic and in fact it shifts (skips) scanning areas in which the algorithm concludes that

no pattern starts at.

As a preliminary step, we present an improved version for the MWM algorithm (see Section III). The modification improves both time and space aspects to fit the large number of patterns within current pattern-sets such as Snort database [6]. We then present Shift-based Pattern matching for Compressed traffic algorithm, *SPC*, that accelerates MWM on compressed traffic. SPC results in a simpler algorithm, with higher throughput and lower storage overhead than the accelerated AC, since MWM basic operation involves shifting (skipping) some of the traffic. Thus, it is natural to combine MWM with the idea of shifting (skipping) parts of pointers.

We show in Section V that we can skip scanning up to 87.5% of the data and gain performance boost of more than 73% as compared to the MWM algorithm on real web traffic and security-tools signatures. Furthermore, we show that the suggested algorithm also gains a normalized throughput improvement of 51% as compared to best prior art [1]. The SPC algorithm also reduces the additional space required for previous scan results by half, by storing only 4KB per connection as compared to the 8KB of [1].

II. Background

Compressed HTTP: HTTP 1.1 [7] supports the usage of content-codings to allow a document to be compressed. The RFC suggests three content-codings: GZIP, COMPRESS and DEFLATE. In fact, GZIP uses DEFLATE with an additional thin shell of meta-data. For the purpose of this paper, both algorithms are considered the same. These are the common codings supported by browsers and web servers.[1]

The GZIP algorithm uses a combination of the following compression techniques: first the text is compressed with the LZ77 algorithm and then the output is compressed with the Huffman coding. Let us elaborate on the two algorithms:

(1) LZ77 Compression [8]- which reduces the *string presentation size* by spotting repeated strings within the last 32KB of the uncompressed data. The algorithm replaces each repeated string by (*distance*,*length*) pair, where *distance* is a number in [1,32768] (32K) indicating the distance in bytes of the repeated string from the current pointer location and *length* is a number in [3,258] indicating length. For example, the text: 'abcdefgabcde' can be compressed to: 'abcdefg(7,5)'; namely, "go back 7 bytes and copy 5 bytes from that point". LZ77 refers to the above pair as "pointer" and to uncompressed bytes as "literals".

(2) Huffman Coding [9]- Recall that the second stage of GZIP is the Huffman coding, that receives the LZ77 symbols as input. The purpose of Huffman coding is to reduce the *symbol coding size* by encoding frequent symbols with fewer bits. Huffman coding assigns a variable-size *codeword* to symbols. *Dictionaries* are provided to facilitate the translation of binary codewords to bytes.

[1]Analyzing packets from Internet Explorer, FireFox and Chrome browsers shows that they accept only the GZIP and DEFLATE codings.

Deep packet inspection (DPI): Essential to almost every intrusion detection system is the ability to search through packets and identify content that matches known attacks. Space and time efficient string matching algorithms are therefore important for inspection at line rate. The two fundamental paradigms to perform string matching derive from deterministic finite automaton (DFA) based algorithms and shift-based algorithms. The fundamental algorithm of the first paradigm is Aho-Corasick (AC) [4], which provides deterministic linear time in the size of the input. The most popular algorithm of the second paradigm is the the modified Wu-Manber (MWM) [5]. The algorithm does not have a deterministic performance, hence it may be exposed to algorithmic attacks. Still, such attacks can be easily identified and the system can switch to using another engine with deterministic characteristics. Overall, the average case performance of MWM is among the best of all multi-pattern string matching algorithms.

A. The Challenges of Performing DPI over Compressed Traffic

Recall that in the LZ77 compression algorithm each symbol is determined dynamically by the data. For instance, string representation depends on whether it is a part of a repeated section and on the distance from that occurrence, which in turn, implies that the LZ77 (and hence, GZIP) is an adaptive compression. Thus, decoding the pattern is futile for DPI, since it will not appear in the text in some specific form, implying that there is no "easy" way to do DPI without decompression. Still, decompression is a considerably light process that imposes only a slight performance penalty on the entire process: LZ77 decompression requires copying consecutive sequences of bytes and therefore benefits from cache advantages gained by spatial and temporal locality. Huffman decoding is also a light task that requires most of the time a single memory access per symbol to a 200B dictionary.

The *space* required for managing multiple-connection environment is also an important issue to tackle. On such environment, the LZ77 32KB window requires a significant *space* penalty since in order to perform deep packet inspection, one needs to maintain 32KB windows of all active compressed sessions. When dealing with thousands of concurrent sessions, this overhead becomes significant. Recent work [10] has shown techniques that circumvents that problem and drastically reduce the space requirement by over 80%, with only a slight increase in time.

III. The Modified Wu-Manber Algorithm

In this section, as a preliminary step to SPC, we present the basic MWM algorithm and an improved version of it. The modifications improve both time and space aspects to fit the large number of patterns within current pattern-sets.

MWM can be thought as an extension for the Boyer-Moore (BM) [11] single-pattern-matching algorithm. In that algorithm, given a single pattern of length n to match, one can look ahead in the input string by n characters. If the character at this position is not a character from our pattern, we can immediately move the search pointer ahead by n characters

without examining the characters in between. If the character appears in the string, but is not the last character in the search string, we can skip ahead by the largest number of bytes that ensures that we have not missed an instance of our pattern. This technique is adapted in a straightforward manner to most implementations of shift-based multi-pattern string matching algorithms, including MWM. The algorithm fits to a fixed length pattern, hence MWM trims all patterns to their m bytes prefix, where m is the size of the shortest pattern. In addition, determining shift-value based on a single character does not fit multi-pattern environment since almost every character would appear as the last byte of some pattern. Instead, MWM chooses predefined group of bytes, namely B, to determine the shift value.

Algorithm 1 outlines the main MWM scan loop and the exact pattern match process. MWM starts by precomputing two tables: a skip *shift table* called $ShiftTable$ (a.k.a SHIFT in MWM) and a *patterns hash table*, called $Ptrns$ (a.k.a. PREFIX and HASH in MWM). The *ShiftTable* determines the shift value after each text scan. On average, MWM performs shifts larger than one, hence it skips bytes. The scan is performed using a virtual *scan window* of size m. The shift value is determined by indexing the *ShiftTable* with the B bytes suffix of the *scan window* (Line 3). As opposed to MWM that implemented *ShiftTable* as a complete array with all possible keys (i.e., $(|\Sigma|^B)$ where $|\Sigma|$ is the alphabet size), we implement *ShiftTable* as a hash table and store only keys with shift value smaller than the maximal one.

Algorithm 1 The MWM Algorithm

$trf_1 \cdots trf_n$ - the input traffic
pos - the position of the next m-bytes scan window
$ShiftTable$ - array that holds the shift value for each last B-bytes of the window
$Ptrns$ - the pattern-set hashed by the first m-byte of the patterns

```
1:  procedure ScanText(trf_1 ... trf_n)
2:  pos ← 1
3:  while pos + m ≤ n do ▷ Get shiftValue using last B bytes
4:      shiftValue ← ShiftTable[trf_{pos+m−B} ... trf_{pos+m}]
5:      if shiftValue = 0 then          ▷ Check for Exact Matches
6:          for all pat in Ptrns[trf_pos ... trf_{pos+m}] do
7:              if pat = trf_pos ... trf_{pos+pat.len} then
8:                  Handle Match Found
9:              end if
10:         end for
11:         pos ← pos + 1
12:     else
13:         pos ← pos + shftValue                ▷ shftValue > 0
14:     end if
15: end while
```

ShiftTable values determine how far we can shift forward the text scan. Let $X = X_1 \ldots X_B$ be the B-byte suffix of *scan window*. If X does not appear as a substring in any pattern, we can make the maximal shift, $m - B + 1$ bytes. Otherwise, we find the rightmost occurrence of X in any of the patterns: assume that $[q - B + 1, q]$ is the rightmost occurrence of X at any of the patterns. In such a case, we skip $m - q$ bytes. Generally, the values in the *shift table* are the largest possible safe values of skip.

When the *shift table* returns with a 0 value (no shift), a possible match is found. In this case, all m-bytes of *scan window* are indexed into the *Ptrns* hash table to find a list of possible matching patterns. These patterns are compared to the text to find any matches (Lines 6–11). Then the input is shifted ahead by one byte and the scan process continues.

The $Ptrns$ hash-table has a major effect on the performance of MWM. In the original MWM implementation, the *Pattern-Set* is hashed with only B-bytes prefix of the patterns, resulting in an unbalanced hash with long chains of patterns that share the same hash key. For example when $B = 2$, the average chain length is 4.2 for Snort DB, slowing down the exact matching process, where one iterates over all possible pattern-match list and compare each of these patterns to the traffic text. Since the number of patterns grew tremendously in the past years, a longer hash-key should be used, thus we take the entire *scan window* as the hash-key. That reduces the average hash load to 1.44 for Snort DB.

Fig. 2 shows an excerpt of the MWM data structure for $B = 2$. All patterns are trimmed to $m = 5$ bytes (Fig. 2(a)). Fig. 2(b) presents *shift table* entries with shift values smaller than the maximal shift. The rest of the byte pairs, not shown in the example, are those which gain the highest shift value of $m - B + 1 = 4$. Byte pairs in the middle of the strings have reduced shift values, and those that are at the end of the strings, such as 'nb' or 'er' with shift value $= 0$, must be checked for exact match. Fig. 2(c) shows an MWM scan example. The *scan window* of length 5 starts at the beginning of the text and advance by skipping segments of the text. Note that most of the time the *scan window* gains shift value larger than 1. There are two cases where the shift value is 0 and the $Ptrns$ hash-table is being queried. The first case returns a $Match$ of the string 'river', while the second does not locate any matched pattern.

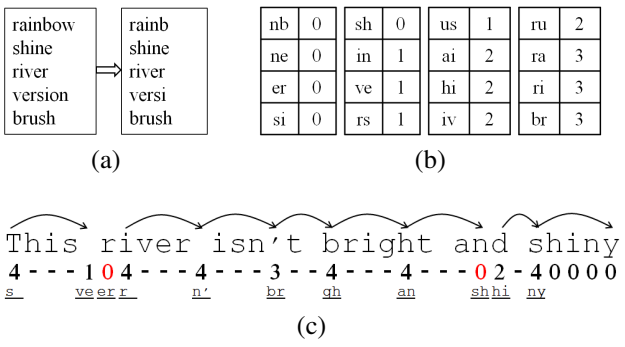


Fig. 2. (a) Pattern set and the m-bytes prefixes. (b) *shift table* of the corresponding pattern-set. (c) MWM scan example. The arrows indicate shifts of the *scan window* larger than 1. The row below the text shows the shift value for each m-bytes *scan window* step. The bottom line contains B-bytes value for shift calculation after each step.

IV. Shift-based Pattern matching for Compressed traffic (SPC)

In this section, we present our *Shift-based Pattern matching algorithm for Compressed HTTP traffic (SPC)*. Recall that HTTP uses GZIP which, in turn, uses LZ77 that compresses data with pointers to past occurrences of strings. Thus, the bytes referred by the pointers (called *referred bytes* or *referred area*) were already scanned; hence, if we have a prior knowledge that an area does not contain patterns, we can skip scanning most of it.

Observe that even if no patterns were found when the referred area was scanned, patterns may occur in the boundaries of the pointer: a prefix of the referred bytes may be a suffix of a pattern that started previously to the pointer; or a suffix of the referred bytes may be a prefix of a pattern that continues after the pointer (as shown in Fig. 3). Therefore, special care need to be taken to handle pointer boundaries correctly and to maintain MWM characteristics while skipping data represented by LZ77 pointers.

The general method of the algorithm is to use a combined technique that scans uncompressed portions of the data using MWM and skips scanning most of the data represented by the LZ77 pointers. Note that scanning is performed on decompressed data such that both decompression and scanning tasks are performed on-the-fly, while using the pointer information to accelerate scanning. For simplicity and clarity of the algorithm description, the pseudocode is written such that all uncompressed text and previous scan information are kept in memory. However in real life implementation it is enough to store only the last 32KB of the uncompressed traffic.

The SPC pseudocode is given in Algorithm 2. The key idea is that we store additional information of partial matches found within previously scanned text, in a bit-vector called $PartialMatch$. The j^{th} bit is set to $true$ if in position j the m-bytes of the *scan window* match an m-byte prefix of a pattern. Note that we store *partial match* rather than *exact match* information. Hence, if the body of the referred area contains no *partial matches* we can skip checking the pointer's internal. However, if the referred area contains a *partial match*, we still need to perform an *exact match*. Maintaining *partial match* rather than *exact match* information is significantly less complicated, especially over skipped characters, due to the fact that pointers can copy parts of the patterns.

The algorithm with the $PartialMatch$ bit-vector integrates smoothly with MWM. In fact, as long as *scan window* is not fully contained within a pointer boundaries, a regular MWM scan is performed (Lines 22–34). The only change is that we update the $PartialMatch$ data structure (Line 25). Note that $shftValue = 0$ implies that the B-bytes suffix of *scan window* matched an entry within $ShiftTable$. It does not imply a *partial match*, which is explicitly checked by querying the $Ptrns$ table.

In the second case, where the m-bytes *scan window* shifts into a position such that it is fully contained within pointer boundaries, SPC checks which areas of the pointer can be skipped (Lines 6–20). We start by checking whether any *partial match* occurred within referred bytes by calling function $findPartialMatches(start \dots end)$ (Line 8). In the simple case where no *partial matches* were found, we can safely shift the *scan window* to $m-1$ bytes before the pointer end (Line 20). In effect we skip the entire pointer body, set the end of the *scan window* one byte passed the pointer and continue with the regular MWM scan. The correctness is due to the fact that any point prior to that point is guaranteed to be free of *partial matches* (otherwise there would have been a match also within the referred bytes). SPC algorithm gains the most

Algorithm 2 The SPC Algorithm

Definitions are as in Algorithm 1, with the following additions:
utrf - The uncompressed HTTP traffic.
pointer - Describes pointer parameters: $distance$,len and $endPos$ - position of the pointer rightmost byte in the uncompressed data. Data received from the decompression phase.
PartialMatch - Bit vector that indicate whether there is a partial match in the already scanned traffic. Each bit in $PartialMatch$ vector has a corresponding byte in $utrf$.
findPartialMatches($start \dots end$) - Returns a list of *Partial Matches* positions in range $start \dots end$.

```
1: procedure ScanText(utrf_{1...n})
2: pos ← 1
3: Set PartialMatch_{1...n} bits to false
4: while pos + m ≤ n do
5:     if utrf_{pos} ... utrf_{pos+m} window internal to pointer then
                                   ▷ Check valid areas for skipping
6:         start ← pos − pointer.dist
7:         end ← pointer.endPos − pointer.dist − (m − 1)
8:         pMatchList ← findPartialMatches(start ... end)
9:         if pMatchList is not empty then
10:            for all pm in pMatchesList do
11:                pos ← pm.pos + pointer.dist
12:                PartialMatch[pos] ← true
13:                for all pat in Ptrns[utrf_{pos} ... utrf_{pos+m}] do
14:                    if pat = utrf_{pos} ... utrf_{pos+pat.len} then
15:                        Handle Match found
16:                    end if
17:                end for
18:            end for
19:        end if
20:        pos ← pointer.endPos − (m − 1)
21:    else          ▷ MWM scan with PartialMatch updating
22:        shftValue = ShiftTable[utrf_{pos+m−B} ... utrf_{pos+m}]

23:        if shftValue = 0 then
24:            if Ptrns[utrf_{pos} ... utrf_{pos+m}] is not empty then
25:                PartialMatch[pos] ← true
26:                for all pat in Ptrns[utrf_{pos} ... utrf_{pos+m}] do
27:                    if pat = utrf_{pos} ... utrf_{pos+pat.len} then
28:                        Handle Match found
29:                    end if
30:                end for
31:            end if
32:            pos ← pos + 1
33:        else pos ← pos + shftValue          ▷ shftValue > 0
34:        end if
35:    end if
36: end while
```

from shifting over the pointer body without the extra overhead of checking $ShiftTable$ and $Ptrns$ in the cases where there are no actual *partial matches*.

If $findPartialMatches(start \dots end)$ returns *partial matches*, we are certain that those were copied entirely from the referred bytes, therefore, we start by setting the corresponding positions within $PartialMatch$ bit-vector to $true$ (Line 12). For each *partial match*, we then query the $Ptrns$ hash-table to check whether an *exact match* occurs, in the same way as in MWM (Lines 13–17).

Fig. 3 demonstrates the SPC algorithm, using the same pattern-set used in Fig. 2. SPC starts with a regular MWM scan. While sanning, SPC locates the m-bytes prefix 'rainb' and mark it as a *partial match* in $PartialMatch$ bit-vector. Note that this m-bytes prefix did not result in an *exact match* with any pattern in the pattern-set. The algorithm continues the MWM scan until the 'shine' prefix is found, marked as a *partial match* and also *exactly matched* to a pattern in the set. Note that at this point we are still not within a pointer, rather we are at the pointer's left boundary. Note that this pointer refers to an area with no *partial matches*. Therefore it scans only the pointer boundaries and skips its internal area. In this example both boundaries are part of a pattern.

Note that the GZIP algorithm maintains the last 32KB of each session. SPC maintains also the $PartialMatch$ bit-vector, i.e. one bit per byte resulting in 4KB or 36KB altogether. Those 36KB can be stored using cyclic buffer, thus re-using also the $PartialMatch$ bits whenever we cycle to the buffer start. Therefore, we cannot rely on the default initialization of those bits and need to add lines that explicitly set the bits to *false*.

Altogether we keep a 36KB of memory per session, which may result in a high memory consumption in a multi-session environment. Note that most of the memory requirement is due to GZIP and is mandatory for any pattern matching on compressed traffic. As mentioned in Section II-A, recent work [10] has shown techniques that save over 80% of the space required. Those techniques can be combined with SPC and reduce the space to around 6KB per session.

The correctness of the algorithm is captured by the following theorem.

Theorem 1: SPC detects all patterns in the decompressed traffic $utrf$.

Sketch of Proof: The proof is by induction on the index of the uncompressed character within the traffic. Assume the algorithm runs correctly until position pos; namely, it finds all pattern occurrences and marks correctly the $PartialMatch$ vector. We now show that the SPC algorithm: 1. finds if there is a pattern in position pos; 2. if it shifts to $pos + shftValue$ there is no patten that starts after $pos + 1$ and prior to $pos + shftValue$; 3. updates correctly the $PartialMatch$ vector.

The correctness relies on the MWM basic property that if a pattern starts at position j then MWM will set *scan window* at position j and the pattern will be located. If *scan window* at position pos is not contained in a pointer then the validity is straightforward from the correctness of MWM. Otherwise, we need to prove that SPC finds all *partial matches* and *exact matches* correctly. The correctness is derived from the induction hypothesis regarding the validity of the $PartialMatch$ vector up to position pos. ∎

V. Experimental Results

In this section, we analyze SPC and the parameters which influence its performance. In addition, we compare its performance to both MWM and ACCH algorithms.

All the experiments were executed on an Intel Core i5 750 processor, with 4 cores running at 2.67GHz and 4GB RAM. Each core has 32KB L1 data and instruction caches and a 256KB dedicated L2 cache. The third-level (L3) cache is larger, at 8MB, and is shared by all four cores.

A. Data Set

The context of this paper is compressed web traffic. Therefore, we collected HTTP pages encoded with GZIP taken from a list constructed from the Alexa website [2] that maintains web traffic metrics and top-site lists. The data set contains 6781 files with a total uncompressed size of 335MB (66MB in its compressed form). The compression ratio is 19.7%. The ratio of bytes represented by pointers is 92.1% and the average pointer length is 16.65B.

B. Pattern Set

Our pattern-sets were gathered from two different sources: ModSecurity [3], an open source web application firewall (WAF) and Snort [6], an open source network intrusion prevention system.

In ModSecurity, we chose the signatures group which applies to HTTP-responses (since only the response is compressed). Patterns containing regular expressions were normalized into several plain patterns. The total number of ModSecurity patterns is 148.

The Snort pattern-set contains 10621 signatures. As opposed to ModSecurity, Snort is not of the web application domain, therefore, it is less applicable for inspecting threats from incoming HTTP traffic. Nevertheless, since Snort is the prominent reference pattern-set in multi-pattern matching papers, we used it to compare the performance of our algorithm to other pattern-matching algorithms. Since HTML pages contain only printable (Base64) characters, there is no need to search for binary patterns, leaving 6837 textual patterns. We also note, that within our data-set, Snort patterns has a significantly high match rate because of patterns such as "http", "href", "ref=", etc. Our data-set contains 11M matches, which accounts for 3.24% of the text. ModSecurity have a modest number of 93K matches, which accounts for 0.026% of the text.

C. SPC Characteristics Analysis

This section explores the various parameters affecting the performance of SPC over *compressed HTTP* and compares it to the MWM running over uncompressed traffic.

Shift-based pattern matching algorithms, and specifically MWM and SPC are sensitive to the shortest pattern length

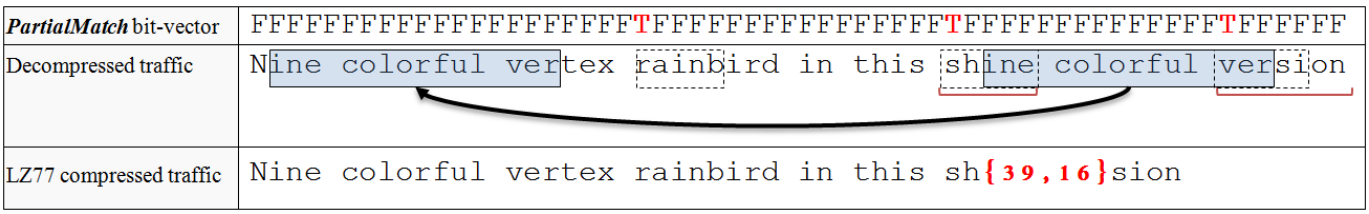


Fig. 3. Pointer scan procedure example. The patterns are as in Fig. 2. The dashed box indicate a *Partial Match* and the solid line indicate an *Exact Match*. The solid box indicate a pointer and its referred area.

as it defines the maximal shift value for the algorithm and influence false positive references to the $Ptrns$ table. It also bounds the size of B, resulting in poor average shift values, since most combinations of those B-bytes are suffixes of our m-prefix patterns. The Snort pattern set contains many short patterns, specifically 410 distinct patterns of length ≤ 3, 539 of length 4 and 381 of length 5. To circumvent this problem we inspected the containing rules. We can eliminate most of the short patterns by using longer pattern within the same rule (as in Snort that defines such pattern with the *fast_pattern* flag) or relying on specific flow parameters (as in [12]). For instance, 74% of the rules that contain these short patterns, contain also longer patterns. Eliminating short patterns is effective for patterns shorter than 5, hence we can safely choose $m = 5$. Still in order to understand the effects of different m and B, we experimented with values for $4 \leq m \leq 6$.

In order to understand the impact of B and m we examined the character of skip ratio, S_r, the percentage of characters the algorithm skips. S_r is a dominant performance factor of both SPC and MWM. Fig. 4 outlines the *skip ratio* as a parameter of m and B and compares the performance of SPC to MWM. As described in Section IV, SPC shift ratio is based on two factors: the MWM shift for scans outside pointers and skipping internal pointer byte scans. When $m = B$, MWM does not skip at all. In that case the SPC shifts are based solely on internal pointer skipping. For $m = B$, S_r ranges from 70% to 60% as m increases; i.e. the factor based on internal pointer skips, is the dominant one for the given m values.

We note that $m = 6$ gains the best performance as it provides the largest *maximal shift* value (equals to $m-B+1$). However, using $m = 6$ as the shortest pattern length discards too many patterns. We chose $m = 5$ as a tradeoff between performance and pattern-set coverage. The *skip ratio* of SPC is much better than that of MWM, and on Snort, for some values of m and B, we get more than twice the *skip ratio*. This property of SPC is a direct result of skipping pointers whose referred bytes were already scanned.

The B parameter determines the text block size on which the $shftValue$ is calculated and has two dominant effects on the performance of MWM and SPC: larger B value decreases the maximal shift, $S_m = m - B + 1$, which correlates directly to the average shift but it also increases part of the shift values as it decreases the percentage of entries which results in shift value is 0. Overall the maximal skip ratio for Snort is 82.7% for $m = 5$ and $B = 3$, whereas on ModSecurity S_r is 87.5% for $m = 5$ and $B = 2$.

D. SPC Run-Time Performance

This section presents the run-time performance of SPC as compared to our improved implementation of MWM (as described in section III) and to ACCH, the only current algorithm that handles compressed web traffic.

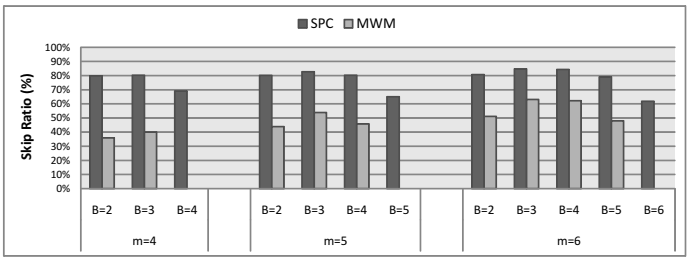
Note that SPC have a basic overhead of 10% over MWM when running on plain uncompressed traffic. This overhead is attributed to the additional $PartialMatch$ bit-vector that impose an overhead of 12.5% memory-references. However since this bit is stored next to its corresponding byte of the traffic and due to the skipping operation we get a smaller overhead.

The algorithms performance is measured by their throughput $T = Work/Time$ (i.e., scanning the entire data-set divided by the scan time). The throughput, as shown in Fig. 5, is normalized to the one of ACCH (which does not depend on m and B values). We note that ACCH's throughput is roughly three times better than Aho-Corasick (AC is omitted from the figure for clarity). ACCH was tuned with optimal parameters as recommended in [1]. The measured throughput of SPC on our experimental environment for Snort is 1.016 Gbit/sec for $m = 5$ and $B = 4$ and for ModSecurity it is 2.458 Gbit/sec for $m = 5$ and $B = 3$. Those results were received by running with 4 threads that performs pattern matching on data loaded in advance to the memory. Our implementation uses C# language and general purpose software libraries and is not optimized for the best throughput. Our goal is to compare between the different algorithms for better understanding of SPC characteristics. Better throughput can be gained by using optimized software libraries or hardware optimized to networking.

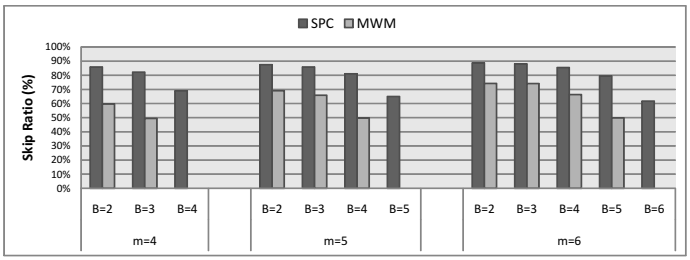
As can be seen, for $m = 5$, when running on Snort, SPC's throughput is better than ACCH's by up to **51.86%**, whereas on ModSecurity, we get throughput improvement of **113.24%**. When comparing SPC to MWM the throughput improvement is **73.23%** on Snort, and **90.04%** on ModSecurity. Note that for all m and B values, SPC is faster that MWM. The maximum throughput is achieved for $m = 5$ when $B = 4$ while the maximum skip ratio is achieved for $B = 3$. This is due to the fact that when B is larger we avoid unnecessary false positive references to the $Ptrns$ data structure. Furthermore, we found out that for the Snort pattern-set we reach a small value of 0.3 memory reference per char.

E. SPC Storage Requirements

We elaborate on the data structures that are used by SPC and MWM: $ShiftTable$ and $Ptrns$ As explained in Section III.

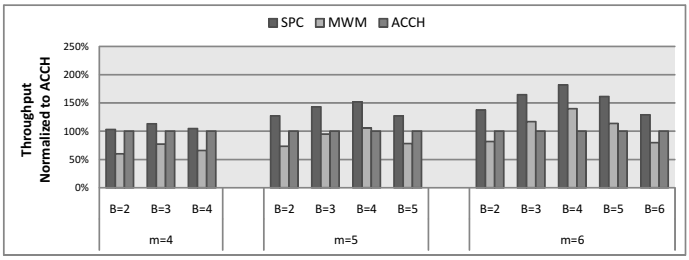


(a). Snort

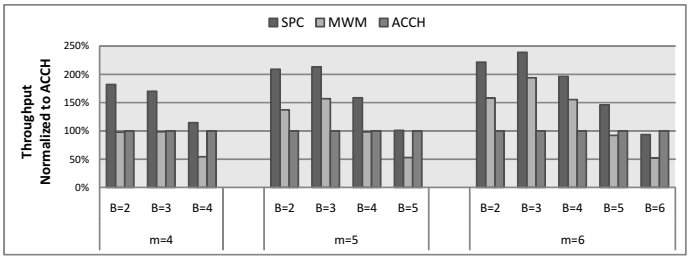


(b). ModSecurity

Fig. 4. Skipped Character Ratio (S_r)



(a). Snort



(b). ModSecurity

Fig. 5. Normalized Throughput

- $ShiftTable$ - is a hash-table that holds shift values and uses the last B-bytes of the *scan window* as the hash-key. If the key is not found in the hash, the maximal shift value, $S_m = m-B+1$, is returned. Each hash-entry contains a pointer to the shift value and the corresponding list of possible B-bytes keys. Hence, the storage required for each entry of $ShiftTable$ is composed of:
  1) Entry pointer - 32bits per entry
  2) Shift value - The maximal value needed to be stored is the maximal shift value-1, hence there are $m-B$ different shift values. To represent each value, we need $log_2(m-B)$ bits
  3) Hash-key - $8 \times B$ bits for each B-bytes key

  The total size of this table is less than 58KB for Snort and less then 1.61KB for ModSecurity.
- $Ptrns$ - is a hash-table of the pattern references (indexes) hashed using the m-bytes pattern prefixes. Each $Ptrns$ table entry holds a pointer to the list of pattern references with the same m-bytes prefix, where each reference is an index to an array which contains the patterns themselves. Hence, for each entry we only need to store:
  1) Entry pointer - 32bits per entry
  2) Pattern index - $log_2(N)$ bits per pattern reference, where N is the number of patterns
  3) Hash-key - $8 \times m$ bits for each m-bytes key

  For Snort, with $m = 5$ this data-structure requires less then $152KB$ whereas, for ModSecurity, it requires only 4.05KB.

Note that we use hash-table implementation such as each hash entry is a list implemented as a fixed size array. This provides a space efficient implementation but is expensive in case of updates to the patterns-set. We believe that for most usage scenarios this is the better tradeoff. However, in case the hash-table needs to support updates, an additional space is needed, as the arrays are replaced by linked lists with pointers, this roughly multiply the memory requirements of the hash-tables by two. Overall, this is still a space efficient implementation.

Table I summarizes the memory required by each of the listed data-structures:

m	B	ShiftTable	Ptrns	Total Storage
Snort				
5	2	14.77	151.73	166.50
5	3	54.35	151.73	206.09
5	4	57.82	151.73	209.55
5	5	36.89	151.73	188.62
ModSecurity				
5	2	1.31	4.05	5.36
5	3	1.61	4.05	5.66
5	4	1.40	4.05	5.45
5	5	0.98	4.05	5.03

TABLE I
STORAGE REQUIREMENTS (KB)

Our implementation is very space efficient as we receive that both MWM and SPC algorithms requires around 1.88 bytes per char for Snort and ModSecurity as opposed to 1.4KB [13] of the original MWM algorithm. This small space requirement increases the probability that the entire table would reside within the cache and thus is a key factor responsible for the performance achieved by the algorithm.

VI. RELATED WORK

Compressed pattern matching has received attention in the context of the Lempel-Ziv compression family [14]–[17].

However, the LZW/LZ78 are more attractive and simple for pattern matching than LZ77. Recall that HTTP uses LZ77 compression, and hence all the above works are not applicable to our case. Klein and Shapira [18] suggested modification to the LZ77 compression algorithm to make the task of the matching easier in files. However, their suggestion is not implemented in today's web traffic.

Farach et. al [19] 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 single pattern and requires two passes over the compressed text (file), which does not comply to the 'on-the-fly' processing requirement applied by the network domains.

The first attempt that tackles the problem of performing efficient pattern matching on compressed HTTP traffic, i.e., on the LZ77 family and in the context of networking is presented in [1]. The paper suggests that the pattern matching task can be accelerated using the compression information. In fact, that paper shows that pattern matching on compressed HTTP traffic, with the overhead of decompression is faster than DFA-based pattern matching (such as Aho-Corasick algorithm [4]). Our paper shows that the same approach can be applied on another important family of pattern matching algorithms, the shift-based technique, such as Boyer-Moore [11] and the modified Wu-Manber (MWM) [5]. We show that accelerating MWM pattern algorithm results in a simpler algorithm, with higher throughput and lower storage overhead than accelerating Aho-Corasick Algorithm. The algorithm can be combined with enhanced solutions based on MWM such as [20]–[25] and also can be implemented for TCAM environment as in [12].

VII. Conclusion and Future Work

With the sharp increase in cellular web surfing, HTTP compression becomes common in todays web traffic. Yet due to its performance requirements, most security devices tend to ignore or bypass the compressed traffic and thus introduce either a security hole or a potential for a denial of service attack. This paper presents *SPC*, a technique that takes advantage of the information within the compressed traffic to accelerate rather than slow down the entire pattern matching process. The algorithm gains a performance boost of over 51% using half the space of the additional information per connection compared to previous known solution, ACCH. The algorithm presented in this paper should encourage vendors to support inspection of such traffic in their security equipment. As for future work we plan on handling regular expression matching over compressed web traffic.

References

[1] A. Bremler-Barr and Y. Koral, "Accelerating multi-patterns matching on compressed HTTP," in *INFOCOM 2009. 28th IEEE International Conference on Computer Communications*, April 2009.

[2] "Top sites," July 2010. http://www.alexa.com/topsites.

[3] "Modsecurity." http://www.modsecurity.org (accessed on July 2008).

[4] A. Aho and M. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, pp. 333–340, 1975.

[5] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Tech. Rep. TR-94-17, Department of Computer Science, University of Arizona, May 1994.

[6] "Snort." http://www.snort.org (accessed on October 2010).

[7] "Hypertext transfer protocol – http/1.1," June 1999. RFC 2616, http://www.ietf.org/rfc/rfc2616.txt.

[8] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, pp. 337– 343, May 1977.

[9] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of IRE*, p. 10981101, 1952.

[10] Y. Afek, A. Bremler-Barr, and Y. Koral, "Efficient processing of multi-connection compressed web traffic," in *IFIP NETWORKING*, 2011.

[11] R. Boyer and J. Moore, "A fast string searching algorithm," *Communications of the ACM*, pp. 762 – 772, October 1977.

[12] Y. Weinsberg, S. Tzur-David, D. Dolev, and T. Anker, "High performance string matching algorithm for a network intrusion prevention system (nips)," in *HPSR*, 2006.

[13] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memoryefficient string matching algorithms for intrusion detection," in *INFOCOM 2004*, 2004.

[14] A. Amir, G. Benson, and M. Farach, "Let sleeping files lie: Pattern matching in z-compressed files," *Journal of Computer and System Sciences*, pp. 299–307, 1996.

[15] 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.

[16] 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.

[17] G. Navarro and J. Tarhio, "Boyer-moore string matching over ziv-lempel compressed text," in *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pp. 166 – 180, 2000.

[18] S. Klein and D. Shapira, "A new compression method for compressed matching," in *Proceedings of data compression conference DCC-2000, Snowbird, Utah*, pp. 400–409, 2000.

[19] M. Farach and M. Thorup, "String matching in lempel-ziv compressed strings," in *27th annual ACM symposium on the theory of computing*, pp. 703–712, 1995.

[20] R. Liu, N. Huang, C. Kao, C. Chen, and C. Chou, "A fast pattern-match engine for network processor-based network intrusion detection system," in *ITCC*, pp. 97–101, 2004.

[21] S. Antonatos, M. Polychronakis, P. Akritidis, K. G. Anagnostakis, and E. P. Markatos, "Piranha: Fast and memory-efficient pattern matching for intrusion detection," in *IFIP Advances in Information and Communication Technology*, pp. 393–408, 2005.

[22] B. Zhang, X. Chen, L. Ping, and Z. Wu, "Address filtering based wu-manber multiple patterns matching algorithm," in *WCSE*, 2009.

[23] Z. Qiang, "An improved multiple patterns matching algorithm for intrusion detection," in *ICIS*, 2010.

[24] Y. Choi, M. Jung, and S. Seo, "L+1-mwm: A fast pattern matching algorithm for high-speed packet filtering," in *INFOCOM*, 2008.

[25] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis, "E^2xB: A domain-specific string matching algorithm for intrusion detection," in *SEC2003*, 2003.