Space Efficient Deep Packet Inspection of Compressed Web Traffic

Yehuda Afek[a], Anat Bremler-Barr[b,1], Yaron Koral[a,*]

[a] *Blavatnik School of Computer Sciences Tel-Aviv University, Israel*
[b] *Computer Science Dept. Interdisciplinary Center, Herzliya, Israel*

Abstract

In this paper we focus on the process of deep packet inspection of compressed web traffic. The major limiting factor in this process imposed by the compression, is the high memory requirements of 32KB per connection. This leads to the requirements of hundreds of megabytes to gigabytes of main memory on a multi-connection setting. We introduce new algorithms and techniques that drastically reduce this space requirement for such bump-in-the-wire devices like security and other content based networking tools. Our proposed scheme improves both space and time performance by almost 80% and over 40% respectively, thus making real-time compressed traffic inspection a viable option for networking devices.

Keywords: pattern matching, compressed http, network security, deep packet inspection

1. Introduction

Compressing HTTP text when transferring pages over the web is in sharp increase motivated mostly by the increase in web surfing over mobile devices. Sites such as Yahoo!, Google, MSN, YouTube, Facebook and others use HTTP compression to enhance the speed of their content download. In Section 7.2 we provide statistics on the percentage of top sites using HTTP Compression. Among the top 1000 most popular sites 66% use HTTP compression (see Figure 5). The standard compression method used by HTTP 1.1 is GZIP.

This sharp increase in HTTP compression presents new challenges to networking devices, such as intrusion-prevention system (IPS), content filtering and web-application firewall (WAF), that inspect the content for security hazards and balancing decisions. Those devices reside between the server and the client and perform *Deep Packet Inspection* (DPI). Upon receiving compressed traffic the networking device needs first to decompress the message in order to inspect its payload. We note that GZIP replaces repeated strings with back-references, denoted as pointers, to their prior occurrence within the last 32KB of the text. Therefore, the decompression process requires a 32KB buffer of the recent decompressed data to keep all possible bytes that might be back-referenced by the pointers, what causes a major *space* penalty. Considering today's mid-range firewalls which are built to support 100K to 200K concurrent connections, keeping a buffer for the 32KB window for each connection occupies few gigabytes of main memory. Decompression causes also a *time* penalty but the time aspect was successfully reduced in [1].

This high memory requirement leaves the vendors and network operators with three bad options: either ignore compressed traffic, or forbid compression, or divert the compressed traffic for offline processing. Obviously neither is acceptable as they present a security hole or serious performance degradation.

The basic structure of our approach is to keep the 32KB buffer of all connections compressed, except for the data of the connection whose packet(s) is now being processed. Upon packet arrival, unpack its connection buffer and process it. One may naïvely suggest to just keep the appropriate amount of original compressed data as it was received. However this approach fails since the buffer would contain recursive pointers to data more than 32KB backwards. Our technique, called "Swap Out-of-boundary Pointers" (*SOP*), packs the buffer's connection by combining recent information from both compressed and uncompressed 32KB buffer to create the new compressed buffer that contains

*Corresponding author
Email addresses: afek@post.tau.ac.il (Yehuda Afek), bremler@idc.ac.il (Anat Bremler-Barr), yaronkor@post.tau.ac.il (Yaron Koral)

[1] Supported by European Research Council (ERC) Starting Grant no. 259085.

pointers that refer only to locations within itself. We show that by employing our technique for DPI on real life data we reduce the space requirement by a factor of 5 with a time penalty of 26%. Notice that while our method modifies the compressed data locally, it is transparent to both the client and the server.

We further design an algorithm that combines our *SOP* technique that reduces space with the ACCH algorithm which was presented in [1] (method that accelerates the pattern matching on compressed HTTP traffic). The combined algorithm achieves an improvement of 42% on the time and 79% on the space requirements. The time-space tradeoff presented by our technique provides the first solution that enables DPI on compressed traffic in wire speed for network devices such as IPS and WAF.

The paper is organized as follows: A background on compressed web traffic and DPI is presented in Section 2. An overview on the related work appears in Section 3. An overview on the challenges in performing DPI on compressed traffic appears in Section 4. In Section 5 we describe our *SOP* algorithm and in Section 6 we present the combined algorithm for the entire DPI process. Section 7 describes the experimental results for the above algorithms and concluding remarks appear in Section 8.

Preliminary abstract of this paper was published in the proceedings of IFIP Networking 2011 [2].

2. Background

In this section we provide background on compressed HTTP and DPI and its time and space requirements. This helps us in explaining the considerations behind the design of our algorithm and is supported by our experimental results described in Section 7.

Compressed HTTP: HTTP 1.1 [3] 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 as its underlying compression protocol. For the purpose of this paper they are considered the same. Currently GZIP and DEFLATE are the common codings supported by current browsers and web servers.[2]

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:

[2]Analyzing captured packets from last versions of both Internet Explorer, FireFox and Chrome browsers shows that accept only the GZIP and DEFLATE codings.

LZ77 Compression [4]- The purpose of LZ77 is to reduce the *string presentation size*, by spotting repeated strings within the last 32KB of the uncompressed data. The algorithm replaces a repeated string by a backward-pointer consisting of a (*distance*,*length*) pair, where *distance* is a number in [1,32768] (32K) indicating the distance in bytes of the string and *length* is a number in [3,258] indicating the length of the repeated string. For example, the text: 'abcdeabc' can be compressed to: 'abcde(5,3)'; namely, "go back 5 bytes and copy 3 bytes from that point". LZ77 refers to the above pair as "pointer" and to uncompressed bytes as "literals".

Huffman Coding [5]- 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. The Huffman coding method builds a dictionary that assigns to symbols from a given alphabet a variable-size *codeword* (coded symbol). The codewords are coded such that no codeword is a prefix of another so the end of each codeword can be easily determined. *Dictionaries* are constructed to facilitate the translation of binary codewords to bytes.

In the case of GZIP, Huffman encodes both literals and pointers. The distance and length parameters are treated as numbers, where each of those numbers is coded with a separate codeword. The Huffman dictionary which states the encoding of each symbol, is usually added to the beginning of the compressed file (otherwise a predefined dictionary is selected).

The Huffman decoding process is relatively fast. A common implementation (cf. zlib [6]) extracts the dictionary, with average size of 200B, into a temporary lookup-table that resides in the cache. Frequent symbols require only one lookup-table reference, while less frequent symbols require two lookup-table references.

Deep packet inspection (DPI): DPI is the process of identifying signatures (patterns or regular expressions) in the packet payload. Today, the performance of security tools is dominated by the speed of the underlying DPI algorithms [7]. The two most common algorithms to perform string matching are the Aho-Corasick (AC) [8] and Boyer-Moore (BM) [9] algorithms. The BM algorithm does not have deterministic time complexity and is prone to denial-of-service attacks using tailored input as discussed in [10]. Therefore the AC algorithm is the standard. The implementations need to deal with thousands of signatures. For example, ClamAV [11] virus-signature database contains 27 000 patterns, and the popular Snort IDS [12] has 6 600 patterns; note that typically the number of patterns considered by IDS systems grows quite rapidly over time.

In Section 6 we provide an algorithm that combines our *SOP* technique with a Aho-Corasick based DPI algorithm. The Aho-Corasick algorithm relies on an underlying Deterministic Finite Automaton (DFA) to support all required patterns. A DFA is represented by a "five-tuple" consisting of a finite set of states, a finite set of input symbols, a transition function that takes as arguments a state and an input symbol and returns a state, a start state and a set of accepting states. In the context of DPI, the sequence of symbols in the input results in a corresponding traversal of the DFA. A transition to an accepting state means that one or more patterns were matched.

In the implementation of the traditional algorithm the DFA requires dozens of megabytes and may even reach gigabytes of memory. The size of the signatures databases dictates not only the memory requirement but also the speed, since it dictates the usage of a slower memory, which is an order-of-magnitude larger DRAM, instead of using a faster one, which is SRAM based. We use that fact later when we compare DPI performance to that of GZIP decompression. That leads to an active research on reducing the memory requirement by compressing the corresponding DFA [10, 13–17]; however, all proposed techniques suggest pure-hardware solutions, which usually incur prohibitive deployment and development cost.

3. Related Work

There is an extensive research on preforming pattern matching on compressed files as in [18–21], but very limited is on compressed traffic. Requirements posed in dealing with compressed traffic are: (1) on-line scanning (1-pass), (2) handling thousands of connections concurrently and (3) working with LZ77 compression algorithm (as oppose to most papers which deal with LZW/LZ78 compressions). To the best of our knowledge, [22, 23] are the only papers that deal with pattern matching over LZ77. However, in those papers the algorithms are for single pattern and require two passes over the compressed text (file), which is not an option in network domains that require 'on-the-fly' processing.

Klein and Shapira [24] suggest a modification to the LZ77 compression algorithm, to change the backward pointer into forward pointers. That modification makes the pattern matching easier in files and may save some of the required space by the 32KB buffer for each connection. However, the suggestion is not implemented in today's HTTP.

The first paper to analyze the obstacles of dealing with compressed traffic is [1], but it only accelerated the pattern matching task on compressed traffic and did not handle the space problem, and it still requires the decompression. We show in Section 6 that our paper can be combined with the techniques of [1] to achieve a fast pattern matching algorithm for compressed traffic, with moderate space requirement. In [25] an algorithm that applies the Wu-Manber [26] multi-patterns matching algorithm on compressed web-traffic is presented. Although here we combine SOP with an algorithm based on Aho-Corasick, only minor modifications are required to combine SOP with the algorithm of [25].

There are techniques developed for "in-place decompression", the main one is LZO [27]. While LZO claims to support decompression without memory overhead it works with files and assumes that the uncompressed data is available. We assume decompression of thousands of concurrent connections on-the-fly, thus what is for free in LZO is considered overhead in our case. Furthermore, while GZIP is considered the standard for web traffic compression, LZO is not supported by any web server or web browser.

4. Challenges in performing DPI on Compressed HTTP

This section provides an overview of the obstacles in performing deep packet inspection (DPI) in compressed web traffic in a multi-connection setting.

While transparent to the end-user, compressed web traffic needs special care by bump-in-the-wire devices that reside between the server and the client and perform DPI. The device needs first to decompress the data in order to inspect its payload since there is no apparent "easy" way to perform DPI 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 symbol is determined dynamically by the data. As a result, the same substring is encoded differently depending on its location within the text. For example the pattern 'abcdef' can be expressed in the compressed data by $abcde *^j (j + 5, 5) f$ for all possible $j < 32763$.

One of the main problems with the decompression is its memory requirement; the straightforward approach requires a 32KB sliding window for each connection. 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 (i.e., a pointer may point to an area with a pointer). As opposed to compressed traffic, DPI of non-compressed traffic requires storing only two or four bytes variable that holds the corresponding DFA state aside of the DFA itself, which is of course stored in any case. Hence, dealing with compressed traffic poses a significantly higher memory requirement by a factor of 8 000 to 16 000. Thus, mid-range firewall that handles 100K-200K concurrent connections (like GTA's G-800 [28], SonicWalls Pro 3060 [29] or Stonesofts StoneGate SG-500 [30]), needs 3GB-6GB memory while a high-end firewall that supports 500K-10M concurrent connections (like the Juniper SRX5800 [31] or the Cisco ASA 5550 or 5580 [32]) would need 15GB-300GB memory only for the task of decompression. This memory requirement not only imposes high price and infeasibility of the architecture but also implies on the capability to perform caching or using fast memory chips such as SRAM. Hence, reducing the space boosts the speed also because faster memory technology is becomming a viable option, such as SRAM memory. This work deals with the challenges imposed by that space aspect.

Apart from the space penalty described above, the decompression stage also increases the overall *time* penalty. However, we note that DPI requires significantly more time than decompression, since decompression is based on reading consecutive memory locations and therefore enjoys the benefit of cache block architecture and has low per-byte read cost, where as DPI employs a very large data structure that is accessed by reads to non-consecutive memory areas therefore requires expansive main memory accesses. In [1] two of us provided an algorithm that takes advantage of information gathered by the decompression phase in order to accelerate the commonly used Aho-Corasick pattern matching algorithm. By doing so, we significantly reduced the time requirement of the entire DPI process on compressed traffic.

5. SOP Packing technique

In this section we describe our packing technique, which reduces the 32KB buffer space requirement to about 5.5KB per connection. The basic idea is to keep all active connection-buffers in their packed form and unpack only the connection under inspection by the DPI process. To achieve this we re-pack the buffer after each packet-inspection completion while keeping a valid updated-buffer as explained below. It has two parts:

- **Packing**: Swap Out-of-boundary Pointers (*SOP*) algorithm for buffer packing.
- **Unpacking**: Our corresponding algorithm for unpacking the buffer.

Whenever a packet is received, the buffer that belongs to the incoming packet connection is unpacked. After the incoming packet processing is finished an updated buffer is packed using the *SOP* algorithm. The next subsections elaborate on these parts of the algorithm.

5.1. Buffer Packing: Swap Out of boundary Pointers (SOP)

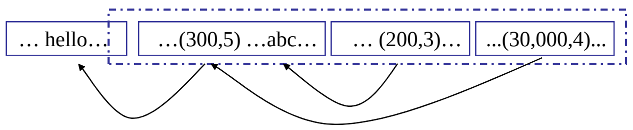
A 1st obvious attempt is to store the buffer in its original compressed form as received on the channel. However this attempt is incorrect since the compressed form of the buffer contains pointers that recursively point to positions prior to the 32KB boundary. Figure 1a shows an example of the original compressed traffic. Note that it contains a pointer to symbols that are no longer within the buffer boundaries. *Clearly, any solution must maintain a valid buffer, which is one that contains only pointers that refer to locations within itself.*

A 2nd obvious attempt is to re-compress (each time from scratch) the 32KB buffer using some compression algorithm such as GZIP. This solution maintains a *valid buffer* as required above, since the compression is based only on information within the buffer. Yet, LZ77 *compression* consumes much more time than *decompression*, usually by a factor of 20. Hence this approach has a very high time penalty, making it irrelevant for real-time inspection. We further show that the space saving by this solution is negligible.

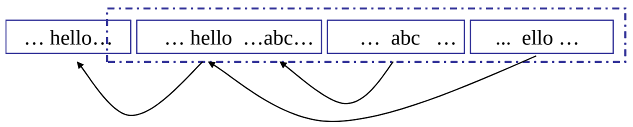
Our suggested solution is called Swap Out-of-boundary Pointers (*SOP*). As in the first attempt, SOP tries to preserve the original structure of the compressed data as much as possible while maintaining the buffer validity. To preserve buffer validity, *SOP* replaces all the pointers that point to location prior the new buffer-boundary with their referred literals.[3] Figure 1c depicts an output of the algorithm. The pointer (300,5) that points outside the buffer boundary is replaced by the string 'hello', where the others remain untouched.

Since in every stage we maintain the invariant that pointers refer only to information within the buffer, the buffer can be unpacked by simply using GZIP decompression as discussed in [33]. *SOP* still has a good compression ratio (as shown in Section 7); most of the pointers are left untouched because they originally pointed to a location within the 32KB buffer boundary. Beside being space efficient, *SOP* is also fast as it performs only single pass on the uncompressed information and

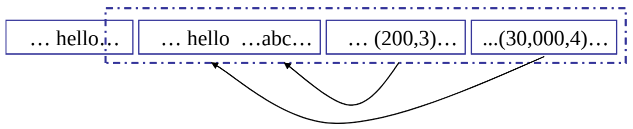
[3]Note that pointers can be recursive, that is pointer A points to pointer B. In that case we replace pointer A by the literals that pointer B points to.



(a) Compressed Buffer



(b) Uncompressed Buffer



(c) SOP Buffer

Figure 1: Sketch of the memory buffer in different scenarios. Each solid box represents a packet. Dashed frame represents the 32KB buffer. Each arrow connects between a pointer and its referred area.

the compressed buffer information, taking advantage of the original GZIP traffic compression that the source (server) has compressed. The pseudocode is given in Algorithm 1. The term *old* buffer refers to the buffer that was packed and stored prior to the new packet arrival and *new* buffer refers to the compressed buffer that is preserved after the processing of the arrived packet. The algorithm consists of 4 parts:

1. Calculation of the arrived packet uncompressed size. This part is required for determining the boundary of the new buffer.
2. Decode the old buffer part which is *outside* the new boundary (its size equals the arrived packet uncompressed size).
3. Decode the old buffer part which is *within* the new boundary.
4. Decode the newly received packet.

The unpacking is divided into three parts (2–4) for code clarity. Recall that the LZ77 pointers and literals are compressed by the Huffman-coding stage. Therefore, the first part performs Huffman decoding of the incoming packet to calculate its uncompressed size which in turn is used to determine the *new boundary*. The *new buffer* contains the arrived packet along with the *old* buffer minus the incoming-packet uncompressed-size. Note that pointers cannot be extracted at this point since they may refer to locations within the old packed buffer which is still compressed.

The other three parts consist of decoding different parts of the data, using a temporary buffer of 32KB uncompressed literals. Part 2 decodes data that would not be re-packed since it is located outside of the new buffer boundary (after receiving the new packet). Parts 3 and 4 decode the rest of the old packed-buffer and the input packet respectively, and prepare the the new packed-buffer along the decoding process. Throughout those parts, each pointer that back-reference to a location outside the new buffer boundary is replaced by its referred literals.

An example of the different parts of the algorithm is illustrated in Figure 1 where the new arriving packet is represented by the rightmost box. The new buffer boundary, which is calculated in *Part 1*, is marked with the dashed box. Notice that part of the leftmost packet does not belong to the new buffer any more, and therefore may not be referred by backward pointers from within the new buffer. On the following parts (2–4) the algorithm replaces each pointer that points to a position prior the new buffer left boundary with its literals, for example the (300,5) in Figure 1.

Notice that in some cases the output is not optimal. Figure 2 depicts a case where LZ77 algorithm (2c) outperforms *SOP* (2d). In that case, the original compression did not indicate any direct connection between the second occurrence of the 'Hello world!' string to the string 'sello world'. The connection can be figured out if one follows the pointers of both strings and finds that they share common referred bytes. Since *SOP* performs a straightforward swapping without following the pointers recursively it misses this case. However, the loss of space is limited as shown in the experimental results Section 7.

Algorithm 1 Out of Boundary Pointer Swapping

packet - input packet.
oldPacked - the old packed buffer received as input. Every cell is either a literal or a pointer.
newPacked - the new packed buffer.
unPacked - temporary array of 32K uncompressed literals.

```
 1: procedure decodeBuffer(buffer)
 2: for all symbols in buffer do
 3:     $S \leftarrow$ next symbol in buffer after Huffman decode
 4:     if $S$ is literal then
 5:         add symbol to unPacked and newPacked buffers
 6:     else                                        ▷ $S$ is Pointer
 7:         store the referred literals in unPacked
 8:         if pointer is out of the boundary then
 9:             store the coded referred literals in newPacked
10:         else
11:             store the coded pointer in newPacked
12:         end if
13:     end if
14: end for

15: procedure handleNewPacket(packet,oldPacked)
    Part 1: calculate packet uncompressed size $n$
16: for all symbols in packet do
17:     $S \leftarrow$ next symbol in packet after Huffman decode
18:     if $S$ is literal then
19:         $n \leftarrow n + 1$
20:     else                                        ▷ $S$ is Pointer
21:         $n \leftarrow n+$ pointer length
22:     end if
23: end for
    Part 2: decode out-of-boundary part of oldPacked
24: while less than $n$ literals were unpacked do
25:     $S \leftarrow$ next symbol in oldPacked after Huffman decode
26:     if $S$ is literal then
27:         store literal in unPacked buffer
28:     else                                        ▷ $S$ is Pointer
29:         store pointer's referred literals in unPacked buffer
30:     end if
31: end while
    Part 3: decode oldPacked part within boundary
32: oldPackedForDecode $\leftarrow$ oldPacked part within boundary
33: if boundary falls within a pointer in oldPacked then
34:     copy to newPacked the referred literals suffix only
35:     include boundary pointer in oldPackedForDecode
36: end if
37: decodeBuffer(oldPackedForDecode)
    Part 4: decode packet
38: decodeBuffer(packet)
```

5.2. Huffman Coding Scheme

So far we discussed only the LZ77 part of GZIP and how to use it for packing each connection-buffer. However, additional space can be saved by using an appropriate symbol-coding such as Huffman coding. In fact some coding scheme must be defined since the LZ77 consists of more than 256 symbols; therefore a symbol cannot be decoded in a straight-forward manner using a single byte. We evaluate three options for symbol coding schemes in our packing algorithm:

1. Original Dictionary - Use the Huffman dictionary that was received at the beginning of the connection.
2. Global Dictionary - Use a global fixed Huffman dictionary for all connections.
3. Fixed Size Coding - Use a coding with a fixed size for all symbols, not based on the Huffman coding.

The first option uses the Huffman dictionary obtained from the *original* connection. Recall that Huffman codes frequent symbols with shorter ones. Notice that *SOP* changes the symbol frequencies therefore the original dictionary is not optimal anymore. Generally, *SOP* has more literals than in the original compressed data since it replaces few pointers with literals. We argue that determining the *SOP* symbol frequencies in order to generate a new dictionary is too expensive.

In the second option one *global Huffman dictionary* is used for all the connections. A dictionary based on those frequencies may provide better compression than the *original* dictionary. Therefore we created a common symbol frequency table using a different training data-set as described in the experimental results section.

Assume we want to avoid the complexity of coding variable size symbols as Huffman does. We suggest a third option and call it the *fixed symbol size* coding scheme. This method assigns a fixed size code for any symbol. The symbol values are based on the way GZIP defines its Huffman alphabets, that is two alphabets for symbol coding: the first alphabet represents the literals and pointer-length values and the second represents the pointer-distance values. Thus there is no need to define a special sign that distincts literals from pointers. The pointer-length value is in [3–258]. Since the literals consist of 256 different possible values a 9 bit fixed length symbol is sufficient to represent all possible values from the first alphabet. The second alphabet requires a 15 bit symbol to represent all possible distances in [1–32768]. If current symbol is a literal the decoder assumes that the next symbol is a 9 bit symbol. Otherwise the next symbol is treated as a 15 bit pointer-distance followed by a 9 bit symbol.

Comparison between the three coding schemes is presented in Section 7. The results show that the best choice is the second option that uses the *global dictionary*. The *original* dictionary option results in a buffer size which is 3% larger and the *fixed size* scheme results in a buffer size which is 7% larger as compared to using the *global dictionary*. Concerning the time requirement, performing Huffman decoding is a light operation, as mentioned in Section 2. The dictionary is small enough to fit in the cache and both coding and decoding consist mostly of single table-lookup operation. Therefore the time differences between the above three options are negligible as compared to the whole process, and result mainly from the different buffer sizes.

5.3. Unpacking the buffer: GZIP decompression

So far we introduced the SOP algorithm which is in charge of buffer packing. We now discuss the complimentary algorithm for unpacking these buffers. SOP buffers are packed in a valid GZIP format, hence a standard GZIP decompression can be used for unpacking.

Still, we should consider the fact that most of the data is decompressed more than once since it is packed and unpacked whenever a connection becomes active upon a new packet arrival. According to our simulations in Section 7, each byte is decompressed on average 4.2 times using *SOP* buffers as compared to only once in the original GZIP method without buffer packing.

Note that upon unpacking we decompress the entire buffer prior to content inspection. One may wonder if we could reduce the above number by extracting only the parts of the buffer that are actually back-referenced by pointers. Since the pointers are recursive, the retrieval of literals referenced by a pointer is a recursive process, which refers to several locations within the buffer. Hence, the number of decoded symbols for a single pointer-extraction may be greater than the pointer length. For example: a pointer that points to another pointer which in turn points to a third pointer, requires decoding three different areas where the referred pointers and the literals reside.

This recursive behavior might lead to referring most of the 32KB buffer, making the cost of maintaining a more complicated technique not worthy. To check this we designed a method for partial decompression, called *SOP-Indexed*. It is defined as follows:

- Split the compressed buffer to fixed size chunks and keep indices that hold starting positions of each chunk within the compressed buffer.
- Recursively extract each pointer and decode only the chunks that are referred by some pointer.

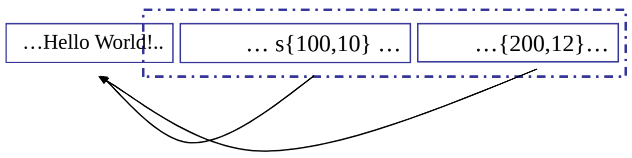
For example: if 256B chunks are used, referring to the 500th byte of the 32KB uncompressed buffer requires decode of the second chunk that corresponds to the [256–511] byte positions within the uncompressed buffer. If the pointer exceeds the chunk boundary of 511, the next chunk has to be decoded too.

Since compressed symbols within the buffer are coded using a variable length number of bits, a chunk might end at a position which is not a multiple of 8. Hence an index to the beginning of the next chunk should indicate a bit offset rather than a byte offset. In order to avoid using bit offset, which requires keeping an eight times larger index, we apply zero padding that pads the end of each chunk with zero bits up to a position that is a multiply of 8. Each index is coded by using 15 bits to represent offsets in [0–32K]. The chunk size poses a time-space tradeoff. Smaller chunks support more precise references and result in less decoding but requires more indices that have to be stored along with the buffer, and more padding for each of the smaller chunks, hence increase the space waste. The analysis in Section 7 shows that a moderate time improvement of 36% as compared to *SOP*, is gained by *SOP-Indexed* along with an average space penalty of 300B per buffer. One may choose whether to apply the *SOP-Indexed* or not according to the specific system needs considering the time-space trade-off that is presented here.

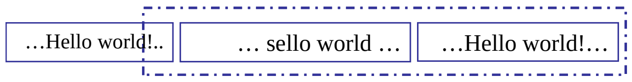
6. Combining SOP with ACCH algorithm

Thus far we discussed SOP, the space efficient buffer packing and unpacking method. In this section we integrate our SOP algorithm with a content inspection algorithm to support the complete DPI process. For that purpose we choose the state-of-the-art ACCH algorithm, presented in [1], which is the first to provide a DPI scheme that inspects GZIP compressed content. The ACCH algorithm gains up to 74% performance improvement as compared to performing DPI with Aho-Corasick (i.e., three times faster). Still, it increases the memory requirement by 25% (8KB per open connection). We suggest a scheme that benefits from the time savings of ACCH and combines with SOP such that it reduces both the 32KB storage requirement of GZIP and the 8KB of ACCH.

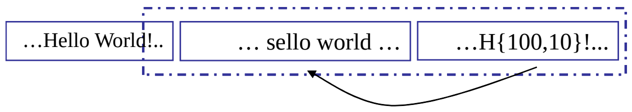
In order to understand the combined SOP-ACCH algorithm, we start with a short overview of ACCH. A detailed description of the algorithm can be found in [1]. The general idea is that: Recall that GZIP compression is done by compressing repeated strings. Whenever the input traffic contains a repeated string represented as an



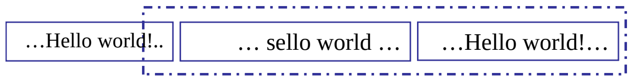
(a) Compressed Buffer



(b) Uncompressed Buffer



(c) Re-compressed Buffer (2^{nd} attempt)



(d) SOP Buffer

Figure 2: Sketch of the memory buffer in different scenarios.

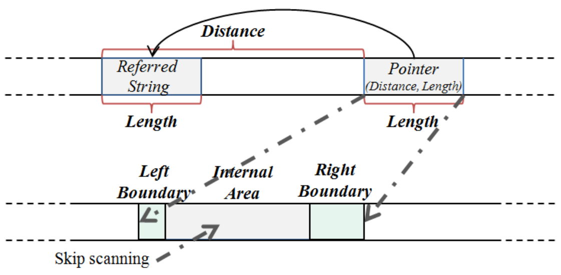


Figure 3: Illustration of *pointer* and its *referred string* with the *left boundary* and *right boundary* scan areas.

LZ77 pointer, ACCH uses the information about previous scans of that string instead of re-scanning it. Since most of the compressed bytes are represented as pointers, most of the byte scans can be saved.

In the general case, if the string that the pointer points to, denoted as the *referred string*, does not completely contain matched pattern then the pointer also contains none. In that case the algorithm may skip scanning bytes in the uncompressed data where the pointer occurs. Apart from that case ACCH handles three special cases where the first two refer to examining whether the string that the pointer represents (which is an identical copy of the *referred string*), denoted by the *pointer string*, is part of a pattern and the third refers to the case where a pattern was found within the referred string. In the first case, the pattern starts prior to the pointer and only its suffix is possibly in the *pointer string*. In the second case the *pointer string* contains a pattern prefix and its remaining bytes occur after the pointer. In order to detect those patterns we need to scan few bytes within the pointer starting and ending points denoted by pointer left boundary and right boundary scan areas respectively (as in Fig. 3). If no matches occurred within the referred string the algorithm skips the remaining bytes between the pointer boundaries denoted by internal area. Note that left boundary, right boundary and internal area are parameters determined by the ACCH algorithm. In the third case a pattern ends within the referred string. The algorithm has to figure out whether the whole pattern is contained within the referred string or only its suffix.

The ACCH algorithm is based on the Aho-Corasick algorithm [8] which uses a DFA for content inspection as defined in Section 2. Handling those three cases relies strongly on using the DFA state *depth* parameter (namely, the number of edges on the shortest simple path from the state and the DFA root). The *depth* of a state is the length of the *longest prefix* of any pattern within the pattern-set [34] at the current scan location. Using the *longest prefix* helps ACCH to determine the locations within the pointer string from where to stop or continue scanning at each one of the above cases. We elaborate on the way ACCH handles those three cases:

Right Boundary (second case): In this case the algorithm determines whether some pattern pattern starts within the pointer suffix, i.e., whether a pattern prefix is in the suffix of the pointer string. Here, ACCH uses the stored information about previous scans in order to determine the *longest prefix* of any pattern within the pattern-set that ends at the last byte of the referred string. ACCH continues its scan at the location of that *longest prefix* within the pointer string which in turn defines the right boundary area.

Match in Referred String (third case): This case is handled similarly to the previous one. Using the information about previous scans, ACCH determines the *longest prefix* of any pattern that ends at the last byte of the matched pattern within the referred string. Knowing that, ACCH determines the point within the pointer string from where to scan in order to check whether the pointer string also contains the pattern which was found while scanning the referred string.

Left Boundary (first case): In this case the algorithm determines whether a pattern ends within the pointer left boundary. The algorithm continue scanning the pointer string (in the uncompressed data) until the number of bytes scanned is equal or greater than the *depth* of the DFA state of the current scanned symbol. At that point the *longest prefix* of any pattern of the pattern-set is contained within the pointer string, thus there cannot be any pattern that started prior to the pattern string and the algorithm may continue to one of the other two cases.

The ACCH algorithm described above increases the memory requirement by 25% (8KB per open connection) in order to maintain the information about previous scans. In order to explain our scheme that reduces this memory requirement we elaborate on the data structure that holds the previous scans information in ACCH, called *Status Vector*.

Recall that ACCH uses the information about previous scans to determine the *longest prefix* in referred strings for the *right boundary* and *match* cases. In fact the vector does not store the exact '*longest-prefix*' length parameter, but whether the prefix length is above or below a certain threshold. Using that threshold allows keeping only two status-indicators instead of storing all possible '*longest prefix*' lengths, which in turn saves space. It is shown in [1] that using this threshold causes only a few extra unnecessary byte scans but does not cause misdetection of patterns. Specifically, each vector entry represents the status of a byte within the previously scanned uncompressed text. The entry is 2-bit long and it stores three possible status values: *Match*, in case that a pattern was located, and two other states; If a prefix longer than a certain threshold was located the status is *Check*, otherwise it is *Uncheck*. This information suffices to get a good estimate of the *longest prefix* that ends at any point within the referred string.

Evidently the 2-bit entry of the *Status Vector* causes the 25% space penalty per byte within the 32KB GZIP window (2 bits for each byte). Our suggested scheme stores the vector in a much more efficient way. Since there are long stretches with the same status, we indicate only a change of status. That is, we place a change-of-status indication after each byte at which such a change occurs. For example, in the case where the *Status Vector* contains 20 *Uncheck* status indicators in a row followed by 3 *Check* status indicators in a row, instead of using 23 pairs of bits as in the ACCH vector, we use only a single symbol that indicates a status change to *Check*, placed in the sequence after the 20 bytes of the input string. For that matter we define three new *Change-Status* symbols in the Huffman dictionary that indicate a status change to each one of the available statuses, namely *Check*, *Uncheck* and *Match*. The start status would always be *Uncheck*, hence it should not be specified. Figure 4d shows an example that demonstrates the usage of *Change-Status*. The *Change-Status* symbols S_u and S_c indicate a change of status, after the string 'hell' to *Uncheck* and after the string 'ab' to *Check* respectively.

Notice that *Change-Status* symbols are applicable only when they can be inserted between two other symbols. That does not apply in the case of a status-change within pointer bytes, since all those bytes are represented by a single *pointer-length* symbol (followed by *pointer-distance* symbol). Here we mark that ACCH is able to restore safely all pointer's statuses from the referred string except for those in the *Left Boundary* (as discussed in [1]). Thus we should only supply a solution that indicates status-changes that occur within the *Left Boundary* bytes.

In Figure 4d the *Left Boundary bits* of the first pointer indicate that there is a status change to *Uncheck* at the first byte and that the *Left Boundary* ends at that point. At the second pointer the *Left Boundary bits* indicate that there was no status change at all and that *Left Boundary* ends after two bytes. The *Left Boundary bits* are encoded by adding a pair of bits per byte in the *Left Boundary*, called *Left Boundary bits*, next to each pointer. The first bit indicates whether it is the last byte of the Left boundary. The second indicates whether a status change occurs within the corresponding byte. In such a case, a third bit indicates to which one of the two the status changes. Note that no special symbol is assigned to those bits within the Huffman dictionary. Instead, we place those bits at the end of each pointer (although those bits hold information about the beginning of the pointer). The *Left Boundary* is rather small and contains only 1.6 bytes on average, thus those

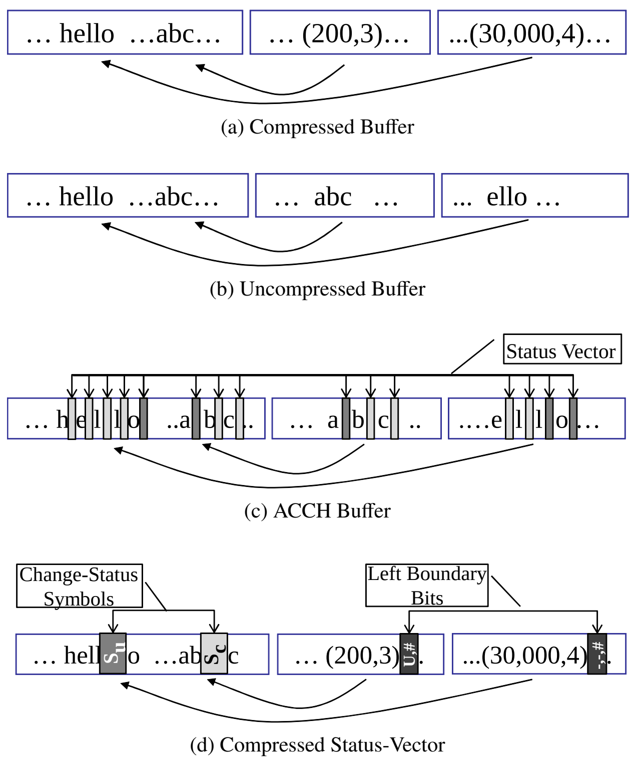


Figure 4: Sketch of the memory buffer including the *Status Vector*. (a) Original compressed traffic (b) Traffic in its uncompressed form (c) *Status Vector* packed with Uncompressed Buffer as in ACCH (d) Compressed *Status Vector*.

Left Boundary bits impose a minor space penalty.

7. Experimental Results

7.1. Experimental Environment

Our performance results are given relative to the performance of a base algorithm *Plain*, which is the decompression algorithm without packing, therefore the processor type is not an important factor for the experiments. The experiments were performed on a PC platform using Intel® Core™2 Duo CPU running at 1.8GHz using 32-bit Operating System. The system uses 1GB of Main Memory (RAM), a 256KB L2 Cache and 2×32KB write-back L1 data cache.

We base our implementation on the *zlib* [6] software library for the compression/decompression parts of the algorithm.

SOP is tested in two simulation setups: *simulated worst-case* setup and *realistic multi-connection* setup. In the *simulated worst-case* setup we flush the entire system's cache upon each packet arrival to create the context-switch effect between multiple connections. Thus every packet results in loading the required buffers and states from main memory. In the *realistic multi-connection* setup an input with interleaved packets from different connections is tested, without flushing the cache between packets arrivals.

7.2. Data Set

The data set consists of 2 308 HTML pages taken from the 500 most popular sites that use GZIP compression. The web site list was constructed from the Alexa site [35], which maintains web traffic metrics and top-site lists. Total size of the uncompressed data is 359MB and in its compressed form it is 61.3MB.

While gathering the data-set from Alexa, we collected statistics about the percentage of the compressed pages among the sites as in Figure 5. The statistics shows high percentage of sites using compression, in particular among the top 1000 sites. As popularity drops the percentage slightly drops. Still, almost one out of every two sites uses compression.

The generation of the *global dictionary* as explained in Subsection 5.2 is based on the LZ77 symbol frequencies in a random sample of 500 compressed pages taken from a corporate firewall (a different data-set).

Packing Method	Symbol Coding Scheme	Average Buffer Size	Space Cost Ratio	Time Penalty
Plain	-	29.9KB	1	1
OrigComp (1st *attempt)*	Original Huffman dict.	4.54KB	0.1518	-
Re-compress (2nd *attempt)*	Original Huffman dict.	5.04KB	0.1576	20.77
SOP	Fixed size	7.33KB	0.245	3.91
SOP	Original Huffman dict.	6.28KB	0.211	3.89
SOP	Global Huffman dict.	5.17KB	0.172	3.85
SOP-Indexed	Global Huffman dict.	5.47KB	0.183	3.49

Table 1: Comparison of Time and Space parameters of different algorithms

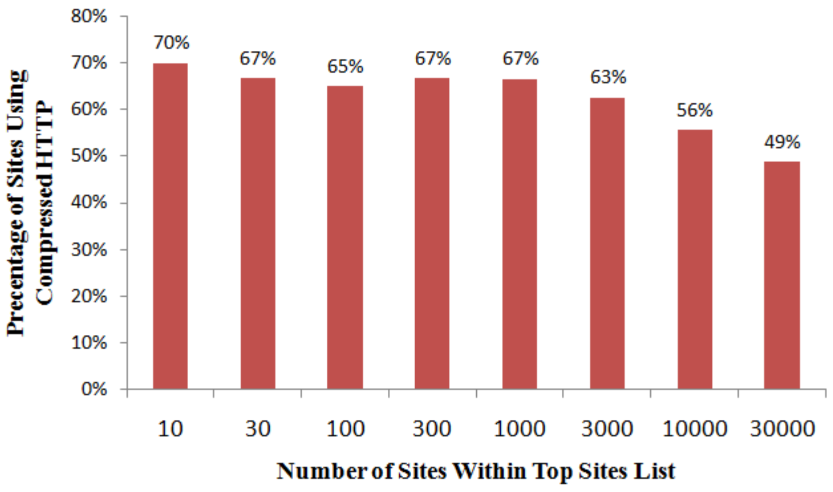


Figure 5: Statistics of HTTP Compression usage among the Alexa [35] top-site lists

7.3. Space and Time Results

This subsection reports space and time results for our algorithm as shown in Table 1. We compare SOP to several *Packing Methods* as described in Section 5 using the three symbol *Coding Schemes* as described in Section 5.2. We define *Plain* as the basic algorithm that performs decompression and maintains a buffer of plain uncompressed data for each connection. The *Average Buffer Size* column indicates the average number of bytes used to store the data of the 32KB window. We are interested in the average ratio rather than in the maximum number of bytes because we try to lower the space requirement in a multi-connection setting, therefore the parameter of a single connection is of less importance. Notice that at the beginning of a connection the buffer stores less than 32KB since it stores only as much data as received. Therefore the average buffer size of *Plain* is 29.9KB which is slightly lower than the maximum value of 32KB. We use *Plain* as a reference for performance comparison to the other proposed methods and set its time and space ratios to 1. The *Space Cost* column indicates the average buffer size of the matching scheme divided by the *Plain* average buffer size. The *Time Penalty* column indicates the time it takes to perform the DPI with the packing and unpacking operations with one method as compared to the same with the *Plain* algorithm.

We measure the size of the incoming compressed data representing the buffer and call it *OrigComp* as the 1st attempt, which is described in Section 5.1. We use this method as a yard stick and a lower bound for the other methods. The average buffer size required by *OrigComp* is 4.54KB.

We define *Re-compress* (2nd attempt), as the method that compresses each buffer from scratch using GZIP. This method represents the best *practical* space result among the compared schemes, but has the worst time requirements of more than 20 times higher than *Plain*.

Next we used the *SOP* packing method with three different symbol coding schemes as described in Section 5.2:

- *Fixed Size coding* scheme.
- *Original* Huffman *dictionary* as maintained from the compressed connection.
- *Global Huffman dictionary*.

The *Global Huffman dictionary* improves the space requirement by a factor of more than 3% compared with the original dictionary and by more than 7% as compared to using the *Fixed Size* coding. This underlines the importance of the Huffman coding part in the space reduction as the literal/pointer ratio grows.

SOP takes 3.85 more time than *Plain*. This is a moderate time penalty as compared to *Re-compress*, the space requirement of SOP is 5.17KB which is pretty close to the 5.04KB of *Re-compress*. The small space advantage gained by *Re-compress* in given its poor time requirement makes it irrelevant as a solution.

We also examined the *SOP-Indexed* method, which maintains indices to chunk offsets within the compressed buffer to support a partial decompression of the required chunks only (see Section 5.2). We used a 256B chunks and got an average of 69% chunk accessed. The time ratio is improved to 3.49 as compared to *Plain*. The space penalty for maintaining the index vector and chunk padding is of 0.3KB.

We simulated *SOP* with the *Global dictionary* using our *realistic multi-connection* mode environment. First we used an input with 10 interleaved connections at a time. The simulations results showed time improvement of 10.5% as compared to the *simulated worst-case* mode. When we used an input with 100 interleaved connections, we got a moderate improvement of 2.8% as compared to the *simulated worst-case* mode.

We simulated *SOP* with the *Global dictionary* using our *realistic multi-connection* setup. Two tests of the realistic setup were carried out, one with 10 concurrent connections and one with 100. The former performed 10.5% faster than the *simulated worst-case* setup and the latter outperformed the worst-case by 2.8%. This behavior is due to the fact that the data of a connection in the cache may be overwritten by the active connection. The more concurrent connections there are the higher the probability that the connections collide in the cache. Thus when a connection becomes active again it has to reload its data into the cache, penalizing its performances. Notice that in the *simulated worst-case* setup we purposely flushed the cache after the processing of each packet.

7.4. Time Results Analysis

As explained in Section 5.3, *SOP* decompresses each byte on average 4.2 times. Hence one would expect *SOP* to take 4.2 times more than *Plain*. Still *SOP* takes only 3.85 times more. This can be explained by inspecting the data structures that require main memory accesses by each of the algorithms (which are responsible to most of the time it takes for the DPI process). *SOP* maintains in main memory the old and new packed buffers (i.e., *oldPacked* and *newPacked* in Algorithm 1) which are heavily accessed during packet processing. *Plain* on the other hand, uses parts of the 32KB buffer, taken from main memory. The memory references made by *Plain* are directly to the 32KB buffer in memory. We measured the relative part of the buffer which is accessed by *Plain* using a method similar to the one in Section 5.3 with chunks of 32B that resembles cache blocks and found out that an average 40.3% of the buffer is accessed. Contrary to *Plain*, *SOP* extracts the 32KB buffer from the old and new packed buffers directly to the cache. Thus, references to this buffer does not cause cache-misses. Only the parts of the 32KB buffer that were replaced by SOP should be written back to main memory. Therefore the penalty of the 4.2 multiple decompressions is lower than expected. Specifically, the average main-memory space used for *SOP* data-structures as compared to 12KB (= .4 of the 32KB) used by *Plain*, which is 20% higher and explains why the time performance of *SOP* is better than the expected 4.2.

7.5. DPI of Compressed Traffic

In this subsection we analyze the performance of the combined DPI process. We focus on pattern matching, which is a lighter DPI task than the regular expression matching. We show that the processing time taken by *SOP* is minor as compared to that taken by the pattern matching task. Since the regular-expression matching task is more expensive than pattern matching, the processing time of *SOP* is even less critical.

Table 2 summarizes the overall time and space requirements for the different methods that implement pattern-matching of compressed traffic in multi-connection environment. The time parameter is relative to the time Aho-Corasick (AC) takes on uncompressed traffic, as implemented by Snort [12]. Recall that AC uses a DFA and basically performs one or two memory references per scanned byte. The other pattern matching algorithm we use for comparison is ACCH [1], which is based on AC and takes advantage from the GZIP compressed input, thus making the pattern matching process

Algorithms in Use			Average Time Ratio	Space per Buffer
Packing	Coding Scheme	Pattern Matching		
Offline	-	AC	-	170KB
Plain	-	AC	1.1	29.9KB
Plain	-	ACCH	0.36	37.4KB
SOP	Global Dict.	AC	1.39	5.17KB
SOP	Global Dict.	ACCH	0.64	6.19KB

Table 2: Overview of Pattern Matching + GZIP processing

faster. However, the techniques used by ACCH are independent from the actual AC implementation. The space per buffer parameter measures the memory required for every connection upon context switch that happens after packet processing is finished, i.e., in *SOP* after packing.

Current network tools that deals with compressed traffic, construct the entire TCP connection first, then they decompress the data and only after that they scan the uncompressed data. We call this method the offline-processing method (*Offline*).The space penalty is calculated by adding the average size of compressed sessions upon TCP reconstruction and is 170KB per connection, which is an enormous space requirement in terms of mid-range security tool.

We use AC processing as the basic time penalty for DPI and normalize it to 1. We measured the average time overhead of the GZIP decompression phase, which precedes the DPI procedure upon each packet arrival and found out that it is 0.101, that is together with the AC the time penalty is 1.101. We note that when performing the same test on a single connection, the GZIP-decompression time overhead is 0.035. The difference is due to the context switch upon each packet on the our setup that harms the decompression spatial locality property.

As explained in Section 6, ACCH improves the time requirement of the pattern matching process. Recall that in order to apply the ACCH algorithm we need to store in memory an additional data structure called *Status Vector*. Applying the suggested compression algorithm to the *Status Vector* as described in Section 6, compressed it to 1.03KB. Therefore the total space requirement of *SOP* combined with ACCH is 6.19KB.

The best time is achieved using *Plain* with ACCH but the space requirement is very high as compared to all other methods apart from *Offline*. Combining *SOP* with ACCH achieves almost 80% space improvement and above 40% time improvement comparing to combining *Plain* with AC.

As we look at the greater picture that involves also DPI, we need to refer to the space requirement applied by its data structures. As noted before, the DPI process itself has a large memory requirement. As opposed to the decompression process whose space requirement is proportional to the number of concurrent connections, the DPI space requirements depend mainly on the number of patterns that it supports. We assume that DPI space requirements are at the same order as those of decompression for mid-range network tools. Thus the 80% space improvement for the decompression buffers translates to a 40% space improvement for the overall process.

8. Conclusions

With the sharp increase in cellular web surfing, HTTP compression becomes common in today web traffic. Yet due to its high memory 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 the *SOP* technique, which drastically reduces this space requirement by over 80% with only a slight increase in the time overhead. It makes realtime compressed traffic inspection a viable option for network devices. We also present an algorithm that combines *SOP* with ACCH, a technique that reduces the time required in DPI of compressed HTTP [1]. The combined technique achieves improvements of about 80% in space and above 40% in the time complexity of the overall DPI processing of compressed web-traffic. Note that the ACCH algorithm (thus the combined algorithm) is not intrusive to the Aho-Corasick (AC) algorithm, and it may be replaced and thus enjoy the benefit of any DFA based algorithm including recent improvements of AC [14, 17, 36].

9. Acknowledgment

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement n°259085.

References

[1] A. Bremler-Barr, Y. Koral, Accelerating multi-patterns matching on compressed http traffic., in: INFOCOM, IEEE, 2009, pp. 397–405.

[2] Y. Afek, A. Bremler-Barr, Y. Koral, Efficient processing of multi-connection compressed web traffic, in: Proceedings of the 10th international IFIP TC 6 conference on Networking - Volume Part I, NETWORKING'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 52–65.

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

[4] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (1977) 337–343.

[5] D. A. Huffman, A method for the construction of minimum-redundancy codes, Proceedings of the Institute of Radio Engineers 40 (1952) 1098–1101.

[6] zlib 1.2.5, April 2010. Http://www.zlib.net.

[7] M. Fisk, G. Varghese, An analysis of fast string matching applied to content-based forwarding and intrusion detection, Techical Report CS2001-0670 (updated version) (2002).

[8] A. V. Aho, M. J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (1975) 333–340.

[9] R. S. Boyer, J. S. Moore, A fast string searching algorithm, Commun. ACM 20 (1977) 762–772.

[10] N. Tuck, T. Sherwood, B. Calder, G. Varghese, Deterministic memory-efficient string matching algorithms for intrusion detection, in: INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, volume 4, pp. 2628 – 2639 vol.4.

[11] Clam antivirus, 2010. Http://www.clamav.net (version 0.82).

[12] Snort, 2010. Http://www.snort.org (accessed on May 2010).

[13] T. Song, W. Zhang, D. Wang, Y. Xue, A memory efficient multiple pattern matching architecture for network security, in: INFOCOM, pp. 166–170.

[14] J. van Lunteren, High-performance pattern-matching for intrusion detection., in: INFOCOM, IEEE, 2006.

[15] V. Dimopoulos, I. Papaefstathiou, D. N. Pnevmatikatos, A memory-efficient reconfigurable aho-corasick fsm implementation for intrusion detection systems., in: H. Blume, G. Gaydadjiev, C. J. Glossner, P. M. W. Knijnenburg (Eds.), ICSAMOS, IEEE, 2007, pp. 186–193.

[16] M. Alicherry, M. Muthuprasanna, V. Kumar, High speed pattern matching for network ids/ips, in: Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols, IEEE Computer Society, Washington, DC, USA, 2006, pp. 187–196.

[17] L. Tan, T. Sherwood, Architectures for bit-split string scanning in intrusion detection, IEEE Micro 26 (2006) 110–117.

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

[19] T. Kida, M. Takeda, A. Shinohara, S. Arikawa, Shift-and approach to pattern matching in lzw compressed text, in: 10th Annual Symposium on Combinatorial Pattern Matching (CPM 99).

[20] G. Navarro, M. Raffinot, A general practical approach to pattern matching over ziv-lempel compressed text, in: 10th Annual Symposium on Combinatorial Pattern Matching (CPM 99).

[21] G. Navarro, 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.

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

[23] L. Gasieniec, M. Karpinski, W. Plandowski, W. Rytter, Efficient algorithms for lempel-ziv encoding (extended abstract), in: In Proc. 4th Scandinavian Workshop on Algorithm Theory, SpringerVerlag, 1996, pp. 392–403.

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

[25] A. Bremler-Barr, Y. Koral, V. Zigdon, Multi-pattern matching in compressed communication traffic, in: Workshop on High Performance Switching and Routing (HPSR 2011), Cartagena, Spain.

[26] S. Wu, U. Manber, A fast algorithm for multi-pattern searching, Technical Report TR94-17 (May 1994).

[27] M. F. Oberhumer, LZO, April 2010. Http://www.oberhumer.com/opensource/lzo.

[28] GB-800/GB-800e DataSheet, 2010. Http://www.gta.com.

[29] SonicWALL PRO 3060, 2010. Http://www.sonicwall.com.

[30] StoneGate FW/VPN Appliances, 2010. Http://www.stonesoft.cn.

[31] SRX5800 specification, 2010. Http://www.juniper.net.

[32] Cisco ASA 5500 series, 2010. Http://www.cisco.com.

[33] P. Deutsch, Gzip file format specification, May 1996. RFC 1952, http://www.ietf.org/rfc/rfc1952.txt.

[34] T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson, Introduction to Algorithms, McGraw-Hill Higher Education, 2nd edition, 2001.

[35] Top sites, July 2010. Http://www.alexa.com/topsites.

[36] W. Lin, B. Liu, Pipelined parallel ac-based approach for multi-string matching, in: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems, IEEE Computer Society, Washington, DC, USA, 2008, pp. 665–672.