Efficient Processing of Multi-connection Compressed Web Traffic

Yehuda Afek, Anat Bremler-Barr, Yaron Koral

Efficient Processing of Multi-Connection Compressed Web Traffic

Yehuda Afek[1], Anat Bremler-Barr*[2], Yaron Koral[1] `afek@math.tau.ac.il`, `bremler@idc.ac.il`, and `yaronkor@post.tau.ac.il`

[1] Blavatnik School of Computer Sciences Tel-Aviv University, Israel
[2] Computer Science Dept. Interdisciplinary Center, Herzliya, Israel

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 major limiting factor in this process is the high memory requirements of 32KB per connection that leads to hundreds of megabytes to gigabytes of main memory consumption. This requirement inhibits most devices from handling compressed traffic, which in turn either limits traffic compression or introduces security holes and other dysfunctionalities. In this paper we introduce new algorithms and techniques that drastically reduce this space requirement by over 80%, with only a slight increase in the time overhead, thus making real-time compressed traffic inspection a viable option for network 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 cellular devices. Sites such as Yahoo!, Google, MSN, YouTube, Facebook and others use HTTP compression to enhance the speed of their content download. In Section 6.2 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 3). 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 content 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. The two major performance penalties associated with this process are *time* and

* Supported by European Research Council (ERC) Starting Grant no. 259085

space. The time it takes to decompress a packet is a small fraction of the time it then takes to inspect the packet in most DPI applications [1]. However the space complexity cost of decompression is a major obstacle specially when the device is dealing with hundreds and thousands of concurrent connections. Notice that the techniques presented in [1], while reducing the DPI time requirement, it still uses information within the compression, i.e., decompression is still required.

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

The basic structure of our approach to dealing with the memory problem is to keep the buffers of all the connections compressed, except for the data of the connection whose packet(s) is now being processed. Upon packet arrival, unpack its session 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 pointers to data more than 32KB backwards. Our technique, called *SOP*, packs the buffer of a connection by combining information from both compressed and uncompressed 32KB buffer to create the new compressed buffer that contains pointers that refer only to locations within itself. We show that by using our technique 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 then design an algorithm that combines our *SOP* technique that reduces space with the ACCH algorithm that reduces time complexity. By using the designed algorithm we achieve improvement of 42% of the time and 79% of the space requirements. The time-space tradeoff presented by our technique provides the first solution that enables DPI on compressed traffic in wire speed.

2 Background

Compressed HTTP: HTTP 1.1 [2] 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 they are considered the same. Currently the GZIP and DEFLATE compressions are the common codings supported by current browsers and web servers.[3]

The GZIP algorithm uses 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:

LZ77 Compression [3]- 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 the repeated strings by (*distance*,*length*)

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

pair, where *distance* is a number in [1,32768] (32K) indicating the distance in bytes of the repeated string and *length* is a number in [3,258] indicating the length. 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".

Note that the LZ77 compression is a time consuming task, while the decompression is considerably light process (we use this observation later on as a motivation in the design of our algorithm). Experiments in Section 6 show that compression takes around 20 times more than decompression. Roughly speaking, the basic idea of the compression process goes as follows: at each point within the traffic, LZ77 tries to find the longest string that has already appeared in the text within the previous 32KB (the most recent if there are multiples). If such a string is found, LZ77 replaces the current string with a pointer to that occurrence. If no repetition longer than 2 bytes is found, than these bytes are not compressed. To decompress the traffic, one needs to reveal the referred bytes by the pointers which translates to a simple operation of consecutive memory copying directly from the 32KB buffer. Reading consecutive bytes has low per-byte read cost due to the good spatial locality in the cache, i.e., reading consecutive 32 bytes within a cache line costs one main memory access.

Huffman Coding [4]- 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 assigns to symbols from a given alphabet a variable-size *codeword* (coded symbol). *Dictionaries* are provided to facilitate the translation of binary codewords to bytes.

The Huffman decoding process is relatively fast. Common implementation (cf. zlib [5]) extracts the dictionary, with average size of 200B, into a temporary lookup-table that resides in 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 main action taken to inspect traffic, by 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 [6]. The two fundamental paradigms to perform string matching derive from Aho-Corasick (AC) [7] and Boyer-Moore (BM) [8] algorithms. The BM algorithm does not have deterministic time and is prone to denial-of-service attacks using tailored input. Therefore the AC algorithm is the standard.The implementations need to deal with thousands of signatures. For example, ClamAV [9] virus-signature database consists of 27K patterns, and the popular Snort IDS [10] has 6.6K patterns; note that typically the number of patterns considered by IDS systems grows dramatically over time. Implementation of the traditional algorithm translates to 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 forces the usage of a larger and slower memory on an order-of-magnitude such as DRAM, instead of using a faster one such as SRAM. That leads to an active research of reducing the memory requirement by compressing the corresponding DFA [11–13]; however, all proposed techniques suggest pure-hardware solutions, which usually incur prohibitive deployment and development cost. Still the common case, definitely in a software solution, requires using a very large database of signatures for DPI. Moreover we note that the DPI solutions do not enjoy the spatial locality time boost. Each input byte requires one or two memory reads to different parts of the memory and thus the DPI solutions do not enjoy the benefits of caching.

3 Challenges in performing DPI on Compressed HTTP

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

As noted in [1], 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.

One of the main problems with the decompression is its memory requirement; the straightforward approach requires a 32KB sliding window for each HTTP 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). On the other hand, DPI of non-compressed traffic requires storing only a two (or four) bytes variable that holds the DFA state. Hence, dealing with compressed traffic poses a higher memory requirement by a factor of 8 000 to 16 000. Thus, mid-range firewall that handles 100K-200K concurrent sessions needs 3GB-6GB memory while a high-end firewall that supports 500K-10M concurrent sessions needs 15GB-300GB memory only for the task of session decompression. This memory requirement has implication on not only the price and feasibility of the architecture but also on the capability to perform caching or using fast memory chips such as SRAM. Thus reducing the space has also straight implication on the speed. 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 consecutive memory reading and therefore enjoy the cache block architecture and has low per-byte read cost, where DPI employs a very large DFA that is accessed by reads to non-consecutive memory areas therefore requires main memory accesses. Our experimental results in section 6 show that the decompression is 10 times faster than the DPI process in a multi-connection environment.

4 Related Work

There is an extensive research on preforming pattern matching on compressed-files, but very limited is on compressed traffic. Requirements posed in dealing with compressed traffic are: (1) on-line scanning (1-pass), (2) handling of thousands of sessions concurrently and (3) working with LZ77 compression algorithm (as oppose to most papers which deal with LZW/LZ78 compressions).

[1] is the first paper to analyze the obstacles of dealing with compressed traffic 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 5.3 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.

There are techniques developed for "in-place decompression", the main one is LZO [14]. 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 sessions 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.

5 Packing technique

In this section we describe our packing technique to reduce the 32KB buffer space requirement per session. The basic idea is to keep the session buffer in its packed form until the time a new incoming packet arrives for that session. To achieve that we use packing technique to keep a correct updated buffer after each packet processing. It has two parts:

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

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

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

In this subsection we describe our buffer packing technique. The first obvious attempt is to store the buffer in its compressed form using the original received traffic. However this attempt fails since the compressed form of the buffer contains pointers that point to positions prior to the 32KB boundary. Figure 1(a) shows an example of the original compressed traffic. Note that it contains pointer to a part that is no longer within the buffer boundaries. The conclusion from this attempt is that the solution must have the following property: A buffer must contain all information for its pointers extraction.

The second obvious attempt is to compress (each time from scratch) the 32KB buffer using some compression algorithm such as GZIP. That solution follows the above property since the compression is based only on information

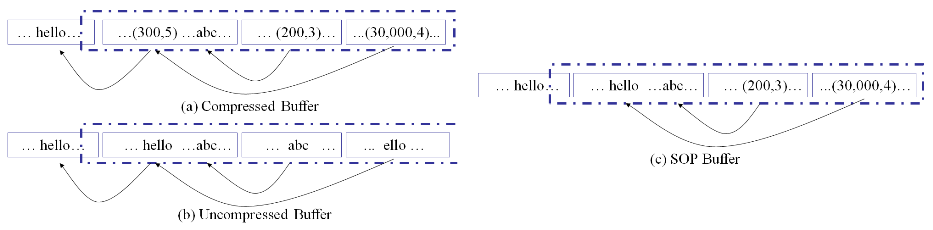

Fig. 1. Sketch of the memory buffer in different scenarios. Each solid box represents a packet. Dashed frame represents the 32KB buffer. Each arrow connects between a pointer and its referred area.

within the buffer. However, this solution performs compression which is an expensive task, while the memory saving is a negligible 1.5% as compared to *SOP*.

Our suggested solution, called Swap Out-of boundary Pointers (*SOP*), solves the problems of the above two attempts. The technique uses information within the original compressed and uncompressed form of the buffer for a quick packing process. *SOP* changes the original GZIP compressed form related to the buffer, so that it contains pointers that refer only to information inside the buffer. To achieve this, *SOP* swaps all the pointers that point outside of the new boundary of the buffer with its referred literals.[4] Figure 1(c) shows an output of this algorithm. The pointer (300,5) that points prior to the buffer new 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. *SOP* still has a good compression ratio (as shown in Section 6) since most of the pointers are left untouched because they originally pointed to a location within the 32KB buffer boundary. *SOP* is also fast since it performs only one pass on the uncompressed information and the compressed buffer information, taking advantage of the GZIP compression that the source (server) did on the traffic.

The pseudocode is given in Algorithm 1. The algorithm consists of 4 parts. The first part performs Huffman decoding of the incoming packet in order to determine its uncompressed size which in turn determines the new boundary of the 32KB buffer. Note that pointers cannot be extracted at this point since *oldPacked* is still compressed. Therefore the decoding is kept within a linked-list data structure that contains either sequence of literals or pointer elements.

The other three parts consist of decoding either *oldPacked* or *packet* into *newPacked* using *unPacked*. Part 2 decodes data that would not be packed again since it is located outside of the new boundary of the 32KB buffer (after receiving the new packet). Parts 3 and 4 decode the rest of *oldPacked* and *packet* respectively, and prepare the *newPacked* buffer along the decoding process.

Note that the output is not optimal in terms of space. Figure 2 shows a case where LZ77 algorithm (2(c)) outperforms *SOP* (2(d)). In that case, the

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

original compression did not indicate of 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 both strings 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.

We also tried some other, more sophisticated variants of *SOP* to solve that problem by some kind of a recursive method. However, the complexity of those algorithms is much higher and the gained space reduction is limited.

5.2 Unpacking the buffer: GZIP decompression

SOP buffers are packed in a valid GZIP format, hence a regular decompression can be used for unpacking. The main difference is that most of the data is decompressed more than once since it is maintained compressed. Buffer decompression is performed upon each incoming packet. Each byte is decompressed on average 4.2 times (see Section 6) using *SOP* buffers as compared to only once in the original GZIP method.

One may wonder if partial decompression of buffer areas is more affordable than the suggested method that decompresses the entire buffer upon each incoming packet. Since the pointers are recursive, the retrieval of the literals referenced by a pointer is a recursive process touching several locations in the buffer, and requiring to decode more symbols than its own 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 property explains the poor results.

We designed a method for partially decompression, called *SOP-Indexed*. It is defined as follows:

- Split the buffer to fixed size chunks of bytes and keep indices that hold starting positions of each chunk.
- Recursively extract each pointer and decode only the chunks that contain required information for extraction.

For example: if a 256B chunks are used, referring to the 500^{th} byte of the 32KB uncompressed buffer requires to decode 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.

Zero padding needs to be applied at the end of each chunk so the offset would be in terms of bytes and not in terms of bits. Each index is coded with 15 bits in order to represent offsets for up to 32KB. The chunk size poses a time-space tradeoff. Smaller chunks support more precise references and result in less decoding but require more indices that have to be stored along with the buffer, and more padding for each of the smaller chunks, hence cause larger space penalty. The results in Section 6 shows that only a limited time improvement of ratio 36% as compared to *SOP*, is gained by *SOP-Indexed*.

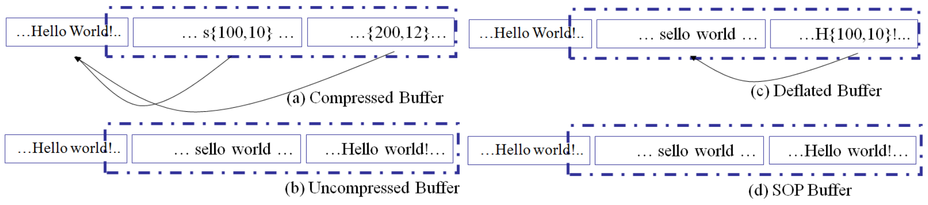

Fig. 2. Sketch of the memory buffer in different scenarios.

5.3 Combining SOP with ACCH algorithm

The algorithm, ACCH presented in [1], reduces the time it takes to do pattern matching to less than 26% compared to doing it with Aho-Corasick (AC). The general idea is to take advantage of pointer information from LZ77 in order to skip and avoid scanning some bytes. Since most of the compressed bytes are represented as pointers, most of the byte scans can be saved. The data about previous byte scans is stored in a *Status Vector* [1]. Each vector entry is 2 bit long and stores three possible state 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*. An important observation is that the status of pointer bytes can be determined based on the status of the referred bytes only, without further scanning.

The ACCH algorithm is somewhat orthogonal to the *SOP* algorithm, still there are two points that must be addressed. The first one is related to the fact that the *Status Vector* itself has 8KB space requirement which needs to be taken care off in order to continue and enjoy the space benefits of *SOP*. The second point is related to the fact that the *SOP* algorithm changes the structure of some of the pointers, therefore when a pointer is replaced by SOP, the vector needs to be efficiently adapted to the new structure.

We use two techniques to address the above points. The first is to mark only the status changes and the second is to use the pointers to figure out the status. We give here a sketch of the suggested method that handles both points due to space limitations. Let us start with the first point. The general idea is to store only status changes instead of the statuses themselves, among the packed buffer symbols. There is no need to store status changes within pointers since ACCH can restore most of the previous referred statuses from the pointer referred area. The rest of the statuses are maintained using extra bits.

Handling the second point means that whenever a pointer is replaced with literals, the symbols that resemble the statuses within it should remain valid. Achieving this is straightforward. The status symbols are maintained from the referred bytes when they are copied, hence no additional memory references are required for status update after pointer replacing.

Applying the combination of the methods described above enabled us to combine ACCH with *SOP* algorithms and gain space and time improvements. The combination achieves time performance of more than two times faster than performing *SOP* with regular AC (as demonstrated in the next section).

6 Experimental Results

6.1 Experimental Environment

Our time 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 2MB L2 Cache and 2 × 32KB write-back L1 data cache.

We base our implementation on the *zlib* [5] software library for the compression/decompression parts of the algorithm.

Our experimental environment does not simulate packets from multiple-connections but only from a single connection at a time. Therefore we flush the entire system's cache upon each packet arrival to create the context switch effect between multiple connections. Note that by flushing the cache we are very conservative in our experimental environment and we suspect that in a real life scenario the time of *SOP* is even better. In the real life environment, some of the writing to the buffers could be performed in the background while handling a different section therefore most of the memory writing penalty could be avoided. Furthermore, sometimes we may receive several consecutive packets from the same flow, hence may be processed without flushing the cache.

6.2 Data Set

The data set consists of 2308 HTML pages taken from the 500 most popular sites that use GZIP compression. The web site list was constructed from the Alexa site [15] that maintains web traffic metrics and top-site lists. Total size of the uncompressed data is 359MB and in its compressed form is 61.3MB.

While gathering the data-set from Alexa, we created an interesting statistics about the percentage of the compressed pages among the sites as shown on Figure 3. The statistics shows high percentage of compression, in particular among the top 1000 sites. As popularity drops the percentage slightly drops. Still, almost 1 out of every 2 sites uses compression.

6.3 Space and Time Results

This subsection reports results concerning the average space requirement for our algorithm as shown in Table 1. We define *Plain* as the basic algorithm that performs decompression and maintains a buffer of plain uncompressed data for each connection. Note that at the beginning of a connection the buffer stores less than 32KB since it stores only as much as was sent. 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.

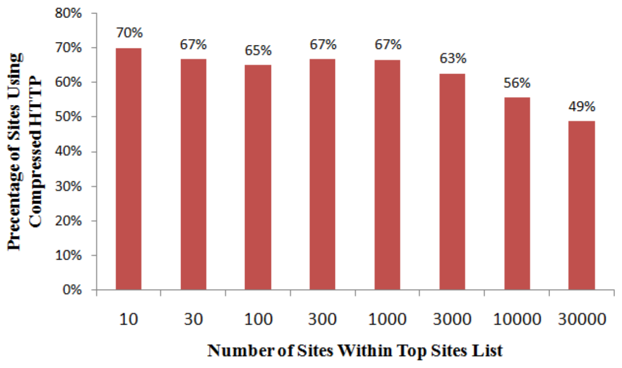

Fig. 3. Statistics of HTTP Compression usage among the Alexa [15] top-site lists

We measure the size of the incoming compressed data representing the buffer and call it *OrigComp*. This data can not be used as a buffer for a *multi-connection environment*, since it contains pointers that point to positions prior to the 32KB buffer range. The average buffer size required by *OrigComp* is 4.54KB, which is considered as a space lower bound.

We define *Deflate* as the method that compresses each buffer from scratch using GZIP. This method represents the best *practical* space result but has the worse time requirements of more than 20 times higher than *Plain*.

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

The last method we examined is the *SOP-Indexed* which maintains indices to chunk offsets within the compressed buffer to support a partial decompression of the required chunks only. 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.

Packing Method	Average Buffer Size	Compression Ratio per Buffer	Average Time Ratio
Plain	29.9KB	1	1
OrigComp	4.54KB	0.1518	-
Deflate	5.04KB	0.1576	20.77
SOP	7.33KB	0.245	3.91
SOP	6.28KB	0.211	3.89
SOP	5.17KB	0.172	3.85
SOP-Indexed	5.47KB	0.183	3.49

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

6.4 Time Results Analysis

As explained in Section 5.2, *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. This can be explained by inspecting the data structures that require main memory accesses by each of the algorithms. *SOP* maintains in main memory the old and new packed buffers (i.e., *oldPacked* and *newPacked* as 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. When processing a packet, *Plain* touches only parts of the 32KB buffer. We measured the relative part of the buffer which is accessed by *Plain* using a method similar to the one in 5.2 with chunks of 32B. An average 40.3% of the buffer is accessed. *SOP* also uses a 32KB buffer for the uncompressed data, but it keeps it only in the cache and most of it is never written back to the main memory. The key point is that a write-back cache is used, hence this buffer remains in cache. The unpacked 32KB buffer is maintained in a temporary variable and when SOP finishes processing the packet that variable is disposed and *not* written back to main memory. However, working in a write-through cache architecture may harm *SOP* performance. The main memory space used for *SOP* data structures is thus around 10KB upon each incoming packet where as the main memory space accessed by *Plain* is around 12KB (= .4 of the 32KB) which is 20% higher and explains why the time performance of *SOP* is better than the expected 4.2.

6.5 DPI of Compressed Traffic

In this subsection we analyze the performance of DPI. 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 compared to the processing time taken by the pattern matching task. Since the regular expression matching task is even more expensive (than pattern matching), the processing time of *SOP* is even less important.

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 compared to performing Aho-Corasick (AC) on uncompressed traffic, as implemented by snort [10]. 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 which is based on AC and uses techniques that are adjusted to GZIP compressed input making the pattern matching process work 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 session upon context switch that happens after packet processing is finished, i.e., in *SOP* after packing.

The offline-processing method (*Offline*) represents the only option supported by current network tools, that deals with compressed traffic. The space requirement is calculated by rough estimate of average compressed and uncompressed

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

Table 2. Overview of Pattern Matching + GZIP processing

size of 27KB and 156KB respectively. The compressed form is stored during session arrival and the uncompressed data is stored during pattern matching phase. When used in combination with AC as used today, it presents an enormous space requirement in terms of mid-range security tool.

We measured the time ratio of performing *Plain* as compared to pattern matching using AC, both on single and multi-connection environment. On the single-connection environment the ratio is as low as 0.035 where on the multi-connection environment the ratio is 0.101. The difference is due to the context switch upon each packet on the multi-connection environment that harms the decompression spatial locality property. Note that *SOP* is compared to performing *Plain* on a multi-connection environment.

As explained in 5.3, ACCH improves the time requirement of the pattern matching process. Recall that in order to apply the ACCH algorithm we need to store on memory an additional data structure called *Status Vector*. Applying the suggested compression algorithm for the *Status Vector* 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*. It occurs that combining *SOP* and ACCH achieves almost 80% space improvement and above 40% time improvement comparing *Plain* with AC.

As we look at the greater picture that involved also DPI, we need to refer to the space requirement applied by its data structures. As noted before, the DPI process too has large memory requirements. As opposed to the decompression process which its space requirement is proportional to the number of concurrent sessions, the DPI space requirements depend mainly on the number of patterns that it supports. Therefore, the DPI large space requirement is applied from the first session. Hence, assuming that DPI space requirements are at the same order as those of decompression for mid-range network tools, the 80% space improvement for the decompression buffers is translated to a 40% space improvement for the overall process.

7 Conclusion

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 *SOP*, a technique that drastically reduces this space requirement (by over 80%) with only a slight increase in the time overhead. It makes real-time 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 performing DPI on compressed HTTP [1]. The combined technique achieves improvements of almost 80% in space and above 40% in the time requirement for the overall DPI processing of compressed web traffic. Note that 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 [11–13].

8 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 and Y. Koral, "Accelerating multi-patterns matching on compressed HTTP," in *INFOCOM 2009*, April 2009.
2. "Hypertext transfer protocol – http/1.1," June 1999. RFC 2616, http://www.ietf.org/rfc/rfc2616.txt.
3. J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, pp. 337– 343, May 1977.
4. D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of IRE*, p. 10981101, 1952.
5. "zlib 1.2.5," April 2010. http://www.zlib.net.
6. M. Fisk and G. Varghese, "An analysis of fast string matching applied to content-based forwarding and intrusion detection," *Techical Report CS2001-0670 (updated version)*, 2002.
7. A. Aho and M. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, pp. 333–340, 1975.
8. R. Boyer and J. Moore, "A fast string searching algorithm," *Communications of the ACM*, pp. 762 – 772, October 1977.
9. "Clam antivirus." http://www.clamav.net (version 0.82).
10. "Snort." http://www.snort.org (accessed on May 2010).
11. W. Lin and B. Liu, "Pipelined parallel ac-based approach for multi-string matching," in *ICPADS*, 2008.
12. J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *INFOCOM*, pp. 1–13, April 2006.
13. L. Tan and T. Sherwood, "Architectures for bit-split string scanning in intrusion detection," *Micro, IEEE*, pp. 110–117, 2006.
14. M. F. Oberhumer, "LZO." http://www.oberhumer.com/opensource/lzo.
15. "Top sites," July 2010. http://www.alexa.com/topsites.

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.
packetAfterHuffmanDecode - contains the Huffman decoded symbols of the input packet.

```
 1: procedure handleNewPacket(packet,oldPacked)
    Part 1: calculate packet uncompressed size $n$
 2: for all symbols in packet do
 3:     $S \leftarrow$ next symbol in packet after Huffman decode
 4:     packetAfterHuffmanDecode $\leftarrow S$
 5:     if $S$ is literal then
 6:         $n \leftarrow n + 1$
 7:     else                                                  ▷ $S$ is Pointer
 8:         $n \leftarrow n+$ pointer length
 9:     end if
10: end for
    Part 2: decode oldPacked part which is out of boundary
11: while less than $n$ literals were unpacked do
12:     $S \leftarrow$ next symbol in oldPacked after Huffman decode
13:     if $S$ is literal then
14:         store literal in unPacked buffer
15:     else                                                  ▷ $S$ is Pointer
16:         store pointer's referred literals in unPacked buffer
17:     end if
18: end while
    Part 3: decode oldPacked part within boundary
19: if boundary falls within a pointer in oldPacked then
20:     copy to newPacked only the suffix of the referred literals
21: end if
22: for all symbols in oldPacked do
23:     $S \leftarrow$ next symbol in oldPacked after Huffman decode
24:     if $S$ is literal then
25:         add symbol to unPacked and newPacked buffers
26:     else                                                  ▷ $S$ is Pointer
27:         store the referred literals in unPacked
28:         if pointer is out of the boundary then
29:             store the coded referred literals in newPacked
30:         else
31:             store the coded pointer in newPacked
32:         end if
33:     end if
34: end for
    Part 4: decode packet
35: for all symbols in packetAfterHuffmanDecode do
36:     $S \leftarrow$ next symbol in packetAfterHuffmanDecode
37:     This part is the same as Lines [24-33]
38: end for
```