Computer Communications 35 (2012) 810–819

Contents lists available at SciVerse ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

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

ARTICLE INFO

Article history:
Available online 28 January 2012

Keywords:
Pattern matching
Compressed http
Network security
Deep packet inspection

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 32 KB 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.

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 Fig. 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 32 KB of the text. Therefore, the decompression process requires a 32 KB 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 100 K to 200 K concurrent connections, keeping a buffer for the 32 KB 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 32 KB 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 32 KB backwards. Our technique, called "Swap Out-of-boundary Pointers" (*SOP*), packs the buffer's connection by combining recent information from both compressed and uncompressed 32 KB buffer to create the new compressed buffer that contains 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

* Corresponding author. Tel.: +972 523903608.
E-mail addresses: afek@post.tau.ac.il (Y. Afek), bremler@idc.ac.il (A. Bremler-Barr), yaronkor@post.tau.ac.il (Y. Koral).

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

0140-3664/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2012.01.017

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:

LZ77 Compression [4] – The purpose of LZ77 is to reduce the *string presentation size*, by spotting repeated strings within the last 32 KB 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 $[1, 32768]$ (32 K) 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 200 B, 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 6600 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 32 KB 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.

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