Decompression-Free Inspection: DPI for Shared Dictionary Compression over HTTP

Anat Bremler-Barr
The Interdisciplinary Center
Hertzelia, Israel
bremler@idc.ac.il

Shimrit Tzur David
The Interdisciplinary Center
Hertzelia, Israel
shimritd@cs.huji.ac.il

David Hay
The Hebrew University
Jerusalem, Israel
dhay@cs.huji.ac.il

Yaron Koral
Tel Aviv University
Tel Aviv, Israel
yaronkor@post.tau.ac.il

***Abstract*— Deep Packet Inspection (DPI) is the most time and resource consuming procedure in contemporary security tools such as Network Intrusion Detection/Prevention System (NIDS/IPS), Web Application Firewall (WAF), or Content Filtering Proxy. DPI consists of inspecting both the packet header and payload and alerting when signatures of malicious software appear in the traffic. These signatures are identified through pattern matching algorithms.**

The portion of compressed traffic of overall Internet traffic is constantly increasing. This paper focuses on traffic compressed using shared dictionary. Unlike traditional compression algorithms, this compression method takes advantage of the inter-response redundancy (e.g., almost the same data is sent over and over again) as in nowadays dynamic Data. Shared Dictionary Compression over HTTP (SDCH), introduced by Google in 2008, is the first algorithm of this type. SDCH works well with other compression algorithm (as Gzip), making it even more appealing. Performing DPI on any compressed traffic is considered hard, therefore today's security tools either do not inspect compressed data, alter HTTP headers to avoid compression, or decompress the traffic before inspecting it.

We present a novel pattern matching algorithm that inspects SDCH-compressed traffic without decompressing it first. Our algorithm relies on offline inspection of the shared dictionary, which is common to all compressed traffic, and marking auxiliary information on it to speed up the online DPI inspection. We show that our algorithm works near the rate of the *compressed traffic*, implying a speed gain of SDCH's compression ratio (which is around 40%). We also discuss how to deal with SDCH compression over Gzip compression, and show how to perform regular expression matching with about the same speed gain.

I. Introduction

Many networking devices inspect the content of packets for security hazards and balancing decisions. These devices reside between the server and the client and perform Deep Packet Inspection (DPI). Using DPI, a device can examine the payload (and possibly also the header) of a packet, searching for protocol non-compliance, viruses, spam, intrusions, or other predefined criteria to decide whether the packet can pass, if it needs to be dropped or be routed to a different destination.

One of the challenges in performing DPI is traffic compression. In order to save bandwidth and to speed up web browsing, most major sites use compression. *W3Techs* published on January 2012 a *ranking breakdown report* [1] which shows that 43.5% of the websites compress their traffic; when focusing on the top $1,000$ sites, a remarkable 84% of the sites compress their traffic. As the majority of this information is dynamic, it does not lend itself to conventional caching technologies. Therefore, compression is a top issue in the Web community.

This research was supported by the European Research Council under the European Unions Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement n^o 259085.

The first generation of compression is intra-file compression, i.e. the file has references to back addresses. Two very common compression methods of the first generation are Gzip [2] and Deflate [3] which have been both developed in the 90's and are very common in HTTP compression. Both methods use combination of the LZ77 algorithm and the Huffman coding. LZ77 Compression [4] reduces the string presentation size by spotting repeated strings within a sliding window of the uncompressed data. The algorithm replaces the repeated strings by (distance, length) pair, where distance indicates the distance in bytes of the repeated string and length indicates the string's length. Huffman Coding [5] receives the LZ77 symbols as input and reduces the symbol coding size by encoding frequent symbols with fewer bits. Gzip or Deflate work well as the compression for each individual response, but in many cases there is a lot of common data shared by a group of pages, namely *inter-response redundancy*. Therefore, compression methods of the next generation are inter-file, where there is one dictionary that can be referenced by several files. An example of a compression method that uses a shared dictionary is SDCH.

Shared Dictionary Compression over HTTP (SDCH) [6] was proposed by Google Inc, thus, Google Chrome (Google's browser) supports it by default. The forecasts, by a report from *Sure Start* [7], are that Chrome will most likely surpass Firefox by the end of 2011 and Microsoft Internet Explorer in 2012. Thus, the spread of SDCH compression should increase to the same degree. Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The Android operating system, was also introduced by Google and it is currently the world's best-selling Smartphone platform with 38.1% market share in the USA [8], [9]. SDCH code appears also in the Android platform and it is likely to be used in the near future. Therefore, a solution for pattern matching on shared dictionary compressed data is essential for this platform as well. SDCH is complement to Gzip or Deflate, i.e. it could be used before applying Gzip. On webpages containing Google search results, the data size reduction when adding SDCH compression before Gzip is about 40% better than Gzip alone.

The idea of the shared dictionary approach is to transmit the data that is common to each response once and after that send only the parts of the response that differ. In SDCH notations, the common data is called the *dictionary* and the differences are stored in a *delta file*. Specifically, a dictionary is composed of the data used by the compression algorithm, as well as metadata describing its scope and lifetime. The scope is specified by the domain and path attributes, thus, a user can download several dictionaries, even from the same server.

Multi-patterns matching on compressed traffic requires two time-consuming phases: traffic decompression and pattern matching. Currently most security tools either do not scan compressed traffic, or they ensure that there will not be compressed traffic by re-writing the HTTP header between the original client and server. The first method harms security and may be the cause to miss-detection of malicious activity, while the second one harms the performance and bandwidth of both client and server. Another possibility is to look for patterns directly on the compressed traffic, however, this option is vulnerable for simple polymorphic attacks, and therefore is not used. The few security tools that handle HTTP compressed traffic, first construct the full page by decompressing it, and then perform signatures scan. Since security tools should operate in the network speed, this option is usually not feasible.

In this paper we present a novel pattern matching algorithm on SDCH. Our algorithm operates in two phases, the *offline* phase and the *online* phase. The offline phase starts when the device gets the dictionary. In this phase the algorithm uses Aho-Corasick [10] pattern matching algorithm (see details in Section II-C) to scan the dictionary for patterns and marks auxiliary information to facilitate the scan of the delta files. Upon receiving the delta file, it is scanned online using Aho-Corasick algorithm. Since the delta file eliminates repetitions of strings using references to the common strings in the dictionary, our algorithm tries to skip these reference, so each plain-text byte is scanned only once (either in the offline or the online phases). We show that we skip up to 99% of the referenced data and gain up to 56% improvement in the performance of the multi-patterns matching algorithm, compared with scanning the plain-text directly.

A. Related Work

String matching algorithm is an essential building block for numerous applications, therefore, it has been extensively studied [11]. Some of the fundamental algorithms are Boyer-Moore [12], which solve the problem for a single pattern and Aho-Corasick [10] and Wu-Manber [13] for multi-patterns. The basic idea of the Boyer-Moore algorithm is that more information is gained by matching patterns from the right than from the left. This allows to heuristically reduce the number of the required comparisons. The Wu-Manber algorithm uses the same observation; however, it provides a solution for the multi-pattern matching problem. The Aho-Corasick algorithm builds a Deterministic Finite Automaton (DFA) based on the patterns. Thereafter, while scanning an input text, the DFA is processed in a single pass.

There are also several works that target the problem of pattern matching on Lempel-Ziv compressed data [14], [15], [16], [17]. Specifically, a solution for Gzip HTTP traffic, called ACCH, was presented in [18]. This solution utilizes the fact that the Gzip compression algorithm works by eliminating repetitions of strings using back-references (pointers) to the repeated strings. ACCH stores information produced by the pattern matching algorithm, for the already scanned uncompressed traffic, and then in case of pointers, it uses this data in order to determine if there is a possibility of finding a match or it can skip scanning this area. This solution shows that pattern matching on Gzip compressed HTTP traffic with the overhead of decompression is faster than performing pattern matching on regular traffic. A similar conclusion regarding files (as opposed to traffic) was previously presented in [19], [20].However, all these algorithms are geared for first-generation compression methods, while there is no pattern matching algorithms for inter-file compression schemes, such as the rapidly-spreading SDCH.

B. Our Contributions

We are the first to address the problem of pattern matching algorithm for shared dictionary compressed traffic. As mentioned, the spread of this approach increases rapidly, thus, a dedicated solution is essential. In addition, we have designed a novel algorithm that scans only a negligible amount of bytes more than once, as our evaluations confirm (see Section V). This is a remarkable result considering the fact that bytes in the dictionary can be referenced multiple times by different positions in one delta file and moreover, by different delta files. SDCH compression ratio is about 44%, implying that 56% of the data is copied from the dictionary. Thus, our single scan implies that our algorithm achieves 56% improvement in performance compared to scanning the plain-text file.

Our algorithm also has low memory consumption. Our algorithm stores only the dictionary being used (along with some auxiliary information per dictionary). In the case of SDCH, since it was developed for web traffic, one dictionary usually supports many connections. In other words, the memory consumption depends on the number of the dictionaries and their sizes and not in the number of connections, which is the case in intra-file compression methods.

Finally, an important contribution is a mechanism to deal with matching regular-expression signatures in SDCH-compressed traffic. Regular expression signatures gain an increasing popularity due to their superior expressibility [21]. We show how to use our algorithm as a building block for regular expression matching. Our experiments show that our regular expression matching mechanism gains a similar 56% boost in performance.

II. BACKGROUND

A. The SDCH Framework

SDCH is a new compression mechanism proposed by Google Inc. In SDCH, a dictionary is downloaded (as a file) by the user agent from the server. The dictionary contains strings which are likely to appear in subsequent HTTP responses. If, for example, the header, footer, JavaScript and CSS are stored in a dictionary possessed by both user agent and server, the server can construct a delta file by substituting these elements with references to the dictionary, and the user agent can reconstruct the original page from the delta file using these references. By substituting dictionary references for repeated elements in HTTP responses, the payload size is reduced and we can save the cross-payload redundancy. In order to use SDCH, the user agent adds the label SDCH in the *Accept-Encoding* field of the HTTP header. The scope of a dictionary is specified by the domain and path attributes, thus, one server may have several dictionaries and the user agent has to have a specific dictionary in order to decompress the server's compressed traffic. If the user agent already has a dictionary from the negotiated server, it adds the dictionary id as a value to the header *Avail-Dictionary*. If the user agent does not have the specific dictionary that was used by the server, the server sends an HTTP response with the header *Get-Dictionary* and the dictionary path; now, the user agent can construct a request to get the dictionary.

We note that our pattern matching can run in a different machine than the server and the client (e.g., in a a security tool that operates as a proxy between them). Since our algorithm needs the correct dictionary, it can force the server to send a response with the *Get-Dictionary* header by deleting the *Avail-Dictionary* field in the client's request.

B. The VCDIFF Compression Algorithm

SDCH encoding is built upon the VCDIFF compression data format. VCDIFF encoding process uses three types of instructions, called delta instructions: ADD, RUN and COPY. ADD(i, str) means to append to the output i bytes, which are specified in parameter str. RUN(i, b) means to append i times the byte b. Finally, COPY(p, x) means that the interval $[p, p+x)$ should be copied from the dictionary (that is, x bytes starting at position p). The delta file contains the list of instructions with their arguments and the dictionary is one long string composed of the characters that can be referenced by the COPY instructions in the delta file. In the rest of the paper, we ignore the RUN instruction since it is barely used and can be replaced with an equivalent ADD for our purposes.

For example, suppose that the dictionary is DBEAACDBCABC, and the delta file is given by the following commands:

1. ADD (3,ABD)
2. COPY (0,5)
3. ADD (1,A)
4. COPY (4,5)
5. ADD (2,AB)
6. COPY (9,3)
7. ADD (4,AACB)
8. COPY (5,3)
9. ADD (1,A)
10. COPY (6,3)

Thus, the plain-text that should be considered is therefore (bolded bytes were copied from the dictionary):

ABD**DBEAA**A**ACDBC**AB**ABC**AACB**CDB**A**DBC**

C. The Aho-Corasick Algorithm and Automaton

Any networking device that is based on DPI uses some sort of a pattern matching algorithm. One of the fundamental approaches is the Aho-Corasick [10] algorithm, which our algorithm uses. The Aho-Corasick algorithm matches multiple patterns simultaneously, by first constructing a Deterministic Finite Automaton (DFA) representing the patterns set, and then, with this DFA on its disposal, processing the text in a single pass.

Specifically, the DFA construction is done in two phases. First, the algorithm builds a trie of the pattern set: All the patterns are added from the root as chains, where each state corresponds to one symbol. When patterns share a common prefix, they also share the corresponding set of states in the trie. The edges of the first phase are called *forward transitions*. In the second phase, *failure transitions* are added to the trie. These edges deal with situations where, given an input symbol b and a state s, there is no forward transition from s using b. In such a case, the DFA should follow the failure transition to some state s' and take a forward transition from there. This process is repeated until a forward transition is found or until the root is reached, leading to possible *failure paths*.

The DFA in Fig. 1 was constructed for patterns set {E, BE, BD, BCD, BCAA, CDBCAB}. Solid black edges are forward transitions while red scattered edges are failure transitions. Let the *label* of a state s, denoted by $L(s)$, be the concatenation of symbols along the path from the root to s. Furthermore, let the depth of a state s be the length of the label $L(s)$. The failure transition from s is always to a state s', whose label $L(s')$ is the longest suffix of $L(s)$ among all other DFA states. This implies the following property of the Aho-Corasick DFA:

Property 1 *If $L(s')$ is a suffix of $L(s)$ then there is a* failure path *(namely, a path comprised only of failure transitions) from state s to state s'.*

The DFA is traversed starting from root. When the traversal goes through an *accepting state*, it indicates that some patterns are a suffix of the input; one of these patterns always corresponds to the label of the accepting state. Formally, we denote by $s.output$ the set of patterns matched by state s; if s is not an accepting state then $s.output = \emptyset$. Finally, we denote by $scan(s, b)$, the AC procedure when reading input symbol b while in state s; namely, transiting to a new state s' after traversing failure transitions and a forward transition as necessary, and reporting matched patterns in case $s'.output \neq \emptyset$. $scan(s, b)$ returns the new state s' as an output. The correctness of the AC algorithm essentially stems from the following simple property:

Property 2 *Let $b_1, \ldots b_n$ be the input, and let $s_1, \ldots, s_n$ be the sequence of states the AC algorithm goes through, after scanning the symbols one by one (starting from the root of the DFA). For any $i \in \{0, \ldots, n\}$, $L(s_i)$ is a suffix of $b_1, \ldots, b_i$; furthermore, it is the longest such suffix among all other states of the DFA.*

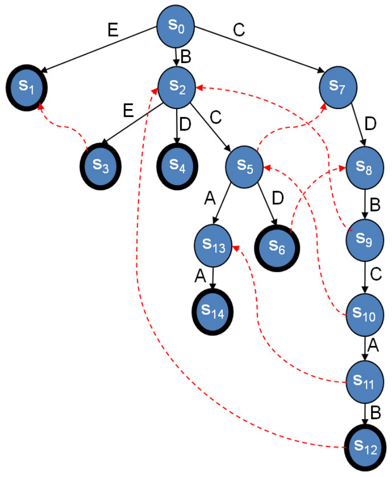

Fig. 1. The Aho-Corasick Automaton corresponding to the pattern set {E, BE, BD, BCD, BCAA, CDBCAB}. Solid black edges correspond to *forward* transitions, while scattered red edges correspond to *failure* transitions.

III. Our Decompression-Free algorithm

A. Motivating Example

We demonstrate the insights behind our algorithm using the following motivating example.

Assume that the patterns set is {E, BE, BD, BCD, BCAA, CDBCAB}, whose corresponding Aho-Corasick automaton is depicted in Fig. 1. In addition, assume the same dictionary and delta file as in the example of Section II-B. The plain-text that should be considered is: ABD**DBEAA**A**ACDBC**AB**ABC**AACB**CDB**A**DBC**, where we marked in bold the symbols that were copied from the dictionary, and in underline patterns that should be matched.

We notice four kinds of matches:

1) Patterns that are fully contained within an ADD instruction. For example, the pattern BD is fully contained within the first instruction.
2) Patterns that are fully contained within a COPY instruction. For example, the pattern BE is fully contained within the second instruction.
3) Patterns whose prefix is within a COPY instruction. For example, the prefix of the pattern CDBCAB is within the fourth instruction.
4) Patterns whose suffix is within a COPY instruction. For example, the suffix of the pattern BCD is within the eighth instruction.

Notice that there might be pattern which fall both in the third and in the fourth category (that is, their prefix is within one COPY instruction, and their suffix is within another COPY instruction).

Our algorithm works in two phases. First, we preprocess the dictionary. Since the dictionary is common to many delta files, this phase runs offline. Then, we process the delta file online. We next describe the two phases, motivating by our example.

The Offline Phase: In the offline phase (Pseudo code in Algorithm 1), the dictionary is scanned *from the first symbol* using the Aho-Corasick algorithm. For each symbol of the dictionary we store the state in which the algorithm was while scanning that symbol. In addition, we keep an ordered list of indices in which a match was found. We will show later that this information is sufficient to skip almost all the symbols of COPY instructions in the delta file. Essentially, this follows from the fact that any scan that starts in the middle of the dictionary will reach states whose labels are suffixes of the states we store (recall Property 2). This, in turn, implies that there is a failure path between the states we store to the corresponding states had the scan was started in the middle of the dictionary (Property 1).

The results of the scan on the above-mentioned example are as follow:

(0) D	(1) B	(2) E	(3) A	(4) A	(5) C	(6) D	(7) B	(8) C
s_0	s_2	$\mathbf{s_3}$	s_0	s_0	s_7	s_8	s_9	s_{10}

(9) A	(10) B	(11) C
s_{11}	$\mathbf{s_{12}}$	s_5

Next we denote by $State[j]$ the state corresponding to the j-th symbol of the dictionary. If the scan reaches an accepting state (that is, a pattern was found within the dictionary), we save it in a list called *Matched*. In that list, we store the index of the last symbol of the matched pattern along with the pattern itself (or, equivalently, a pointer to the pattern). The list is sorted by the index of the symbol. In our example, there are two matches, implying that $Matched = [\langle 2, \{\mathtt{E}, \mathtt{BE}\}\rangle, \langle 10, \{\mathtt{CDBCAB}\}\rangle]$.

The Online Phase: In the online phase (Pseudo code in Algorithm 2), we are scanning the delta file, using the Aho-Corasick algorithm. Since the data in the ADD instruction is new (that is, it was not scanned in the offline phase), we simply scan it by traversing the automaton.

When encountering a COPY (x, p) instruction, which copies the symbols $b_p, \ldots, b_{p+x-1}$ from the dictionary, we are doing the following three steps:

Step 1: Scan the copied symbols from the dictionary one by one, until when scanning a symbol b_{p+i} we reach a state in the automaton whose depth is less or equal to i. As we shall prove later, in this scan we ensure to find all patterns whose suffix is within this COPY instruction (the fourth category of patterns). Notice that the depth of a state essentially indicates the length of meaningful suffix of the input when reaching this state. Therefore, if we reach a state whose depth is less than the number of copied symbols scanned so far, any pattern that ends within the COPY instruction is fully contained in it (the second category). If we reached the end of the copied data (that is, symbol b_{p+x-1}) before encountering such a small depth state, we naturally proceed to the next instruction as all the copied symbol were scanned. Otherwise, we proceed to the next steps.

Step 2: We check the *Matched* list to find any patterns in the dictionary that ends within interval $[x, x+p)$. If such patterns are found, we check by their length that they are indeed fully contained within that interval. Hence, we are ensured to find all the patterns of the second category.

TABLE I

STEP BY STEP EXECUTION OF OUR ALGORITHM ON THE EXAMPLE OF SECTION III-A

1. ADD (3,ABD)	$s_0 \xrightarrow{A} s_0 \xrightarrow{B} s_2 \xrightarrow{D} \mathbf{s_4}$ Report pattern BD (by state s_4)
2. COPY (0,5)	$s_4 \xrightarrow{D} s_0$. Stop scanning copied bytes ($depth(s_0) \le 1$). Check *Matched* for indices $[0,5)$. Report patterns E,BE (from *Matched* list). Go to $State[4] = s_0$. $depth(s_0) \le 5$.
3. ADD (1,A)	$s_0 \xrightarrow{A} s_0$
4. COPY (4,5)	No scanning of copied bytes ($depth(s_0) \le 0$). Check *Matched* for indices $[4,9)$. Go to $State[8] = s_{10}$. $depth(s_{10}) = 4 \le 5$.
5. ADD (2,AB)	$s_{10} \xrightarrow{A} s_{11} \xrightarrow{B} \mathbf{s_{12}}$ Report pattern CDBCAB (by state s_{12})
6. COPY (9,3)	$s_{12} \xrightarrow{A} s_0$. Stop scanning copied bytes ($depth(s_0) \le 1$). Check *Matched* for indices $[9,12)$. Match was found, but does not fit the interval. Go to $State[11] = s_5$. $depth(s_5) = 2 \le 3$.
7. ADD (4,AACB)	$s_5 \xrightarrow{A} s_{13} \xrightarrow{A} \mathbf{s_{14}}$. Report pattern BCAA (by state s_{14}) $s_{14} \xrightarrow{C} s_7 \xrightarrow{B} s_2$.
8. COPY (5,3)	$s_2 \xrightarrow{C} s_5 \xrightarrow{D} \mathbf{s_6}$ Report pattern BCD (by state s_6) $s_6 \xrightarrow{B} s_9$ (reached end of instruction)
9. ADD (1,A)	$s_9 \xrightarrow{A} s_0$
10. COPY (6,3)	No scanning of copied bytes ($depth(s_0) \le 0$). Check *Matched* for indices $[6,9)$. Go to $State[8] = s_{10}$. $depth(s_{10}) = 4 > 3$. $s_{10} \xrightarrow{failure} s_5$. $depth(s_5) = 2 \le 3$.

Step 3: We obtain the state $State[p+x-1]$; namely, the state corresponding to the last copied symbol. From that state, we follow *failure* transitions in the automaton, until we reach a state s whose depth is less or equal to x. (since all *failure paths* in the automaton end in the root whose depth is 0, we are guaranteed to eventually stop). Properties 1 and 2 yield that $L(s)$ is the longest suffix of $b_p, \ldots, b_{p+x-1}$ (among all states' labels). Since we have dealt in Step 1 with all patterns that begin before b_p, the meaningful suffix of the input starts after b_p. This implies that the Aho-Corasick algorithm would have been also in state s, had it scanned all the symbols $b_p, \ldots, b_{p+x-1}$ one by one. Therefore, when processing the next instructions the algorithm behaves exactly the same, guaranteeing to find all patterns of the third category. In addition, this implies identical scans also on ADD instructions, therefore guaranteeing to find all patterns of the first category.

Table I shows a step-by-step execution of our algorithm on the above mentioned delta file. The algorithm reports the same matched patterns, and reaches the same state (s_5) as if it scanned the plain-text. However, the algorithm skips most of the symbols (14 out of 19) within the COPY instructions. We will further investigate the portion of skipped symbol in Section V.

B. Correctness

The pseudo-code of the offline phase is presented in Algorithm 1, while Algorithm 2 depicts the online phase.

Given a dictionary and a delta-file, we denote by $b_1, \ldots, b_n$, the plain-text string obtained by applying the instructions of

Algorithm 1 The Offline Phase

```
1:  $Patterns[] \leftarrow getPatterns(PatternsFile)$
2:  $AC \leftarrow new\ Aho{-}Corasick()$
3:  $AC.CreateTree(Patterns[])$
4:  $Dict \leftarrow loadDictionary(dictFile)$
5:  $cur\_state \leftarrow AC.root$
6:  for $i = 0; i < dict.length; i{+}{+}$ do
7:      $cur\_state \leftarrow AC.scan(cur\_state, dict[i])$
8:      $State[i] \leftarrow cur\_state$
9:      if $cur\_state.output \neq \emptyset$ then
10:         $Matched.addLast(\langle i, cur\_state.output \rangle)$
```

Algorithm 2 The Online Phase

```
1:  $cur\_state \leftarrow AC.root$
2:  while $hasNextInstruction(deltaFile)$ do
3:      $inst \leftarrow getNextInstruction(deltaFile)$
            ▷ $inst$ is a structure containing the instruction along with its parameters
4:      if $inst.type$==ADD then
            ▷ $inst.str$ is the string to append, $inst.len$ is its length
5:          for $i = 0; i < inst.len; i{+}{+}$ do
6:              $cur\_state \leftarrow AC.scan(cur\_state, inst.str[i])$
7:      else if $inst.type$==RUN then
            ▷ $inst.literal$ is the symbol to append, $inst.len$ is the number of times
8:          for $i = 0; i < len; i{+}{+}$ do
9:              $cur\_state \leftarrow AC.scan(cur\_state, inst.literal)$
10:     else                                        ▷ $inst.type$ is COPY.
            ▷ $inst.addr$ is the starting position; $inst.len$ is the length of the copied string
11:         $literals \leftarrow Dict[inst.addr, \ldots, inst.addr{+}inst.len{-}1]$
12:         $i = 0$
13:         while ($cur\_state.depth > i$ and $i \le inst.len$) do
14:             $cur\_state \leftarrow AC.scan(cur\_state, literals[i])$
15:             $i{+}{+}$
16:         if $i \le inst.len$ then
17:             $target\_state \leftarrow State[inst.addr + inst.len - 1]$
18:             while $target\_state.depth > inst.len$ do
19:                 $target\_state \leftarrow target\_state.failure$
                        ▷ the failure transition of $traget\_state$
20:             $cur\_state \leftarrow target\_state$
21:         for all $\langle x, P \rangle \in Matched$ do
22:             if $x \in [inst.addr, inst.addr{+}inst.len{-}1]$ then
23:                 for all $p \in P$ do
24:                     if $x - p.length \ge inst.addr$ then
25:                         report pattern $p$
```

the delta-file. A symbol b_i is *scanned* if Algorithm 2 applies Aho-Corasick *scan* operation to obtain the next state after reading b_i (that is, either in Line 6, Line 9, or Line 14 of Algorithm 2); otherwise, we say that the symbol b_i is *skipped*. Notice that all skipped symbols were copied from the dictionary using a COPY instruction. We further denote by $z_0, z_1, \ldots, z_n$ the states which Aho-Corasick algorithm traverses when running on the plain text string $b_1, \ldots, b_n$ (note that $z_0 = s_0$, the initial state before reading b_1). Finally, we assume, without loss of generality and only for the ease of notations, that the first symbol of a COPY instruction is scanned (this holds anyway, unless the automaton is in state s_0 before the COPY instruction).

Lemma 1 *Before each scanned symbol b_j, Algorithm 2 is in* *state* z_{j-1}.

Proof: For each scanned symbol in $b_j \in \{b_1,\dots,b_n\}$, let $pos(b_j)$ be the number of scanned symbol before b_j. The proof follows by induction on $pos(b_j)$.

Initially, Algorithm 2 is in state $s_0 = z_0$ and the claim follows since the first symbol is scanned. Assume now that the claim holds for all scanned symbols whose *pos* value is less than $pos(b_j)$. We next prove it holds for b_j by distinguishing between the following two cases:

First, if b_{j-1} is a scanned symbol, then, by the induction hypothesis, Algorithm 2 is in state z_{j-2} before reading b_{j-1}. Since b_{j-1} is a scanned symbol, Algorithm 2 applies, by definition, Aho-Corasick *scan* operation to obtain the next state after reading b_{j-1}, and therefore transits to state z_{j-1}, and the claim follows.

Otherwise, the state before reading b_j was set in Line 20 of Algorithm 2, which was executed while processing a COPY instruction. Denote by a the index (in the plain text) of the first symbol that was copied by that COPY instruction; the copied symbols are therefore $b_a,\dots,b_{j-1}$. Notice that Line 20 is reached only if the value of i in Line 16 is less than or equal to $j-a$. This implies that all the symbols $b_a,\dots,b_{a+i-1}$ are scanned, and that the label of the state after scanning b_{a+i-1} is a proper suffix of $b_a,\dots,b_{a+i-1}$ (otherwise the condition in Line 13 does not hold). By the induction hypothesis, since b_{a+i-1} is a scanned symbol that precedes b_j, Algorithm 2 is in state z_{a+i-2} before scanning b_{a+i-1}, and in state z_{a+i-1} after applying the *scan* procedure in Line 14. This implies that $L(z_{a+i-1})$ is a proper suffix of $b_a,\dots,b_{a+i-1}$. Notice that when scanning the $j-(i-a)$ symbols $b_{i+a},\dots,b_{j-1}$, the depth of the state can increase by at most $j-(i-a)$. Hence $L(z_{j-1})$ is a proper suffix of $b_a,\dots,b_{j-1}$; since these symbols are a suffix of the plain text, $L(z_{j-1})$ is the longest such suffix among all states of the automaton (recall Property 2 in Section II-C).

It is important to notice that all these bytes were copied from the dictionary; denote by $c_x,\dots,c_y$ their corresponding indices in the dictionary. Consider now the value of $State[y]$, which obtained by scanning the same DFA from the beginning. Property 2 implies that $L(State[y])$ is the longest suffix, among all DFA's states, of the sequence $c_1,\dots,c_y$, which implies that $L(z_{j-1})$ is a suffix of $L(State[y])$. Thus, by Property 1, there is a path of *failure transitions* on the automaton from $State[y]$ to z_{j-1}; this path is traversed in Lines 18–19. Since $L(z_{j-1})$ is the longest such suffix with length less than $j-a$, the traversal is stopped exactly in z_{j-1}. Thus, Algorithm 2 transits to state z_{j-1} also in this case, concluding our proof. ■

Lemma 2 *All matched patterns that end with a skipped symbol are fully contained within a single* COPY *instruction.*

Proof: Assume towards a contradiction that this is not a case, and consider the other three possibilities.

First, for patterns that are fully contained within an ADD instruction, all symbols of ADD instructions are scanned and therefore these patterns cannot end with a skipped symbol.

Similarly, for patterns whose prefix is within a COPY instruction and suffix is in an ADD instruction, the last symbol of these patterns is always scanned.

Finally, we deal with patterns whose suffix is within a COPY instruction, but are not entirely within that COPY instruction. Consider the symbols $b_a,\dots,b_{j-1}$ which were copied in that COPY instruction, and assume that b_k is the last symbol of the pattern within these symbol ($a \le k < j$). Since the pattern exceeds the copied symbols, it implies that $L(z_k)$ is longer than $k-a$ symbols. Since a depth of successive states can increase only by one, it implies that the depth of all states z_ℓ for $a \le \ell \le k$ is strictly larger than $\ell - a$. This implies that the condition of Line 13 holds for all these states, including b_k. This, in turn, implies that b_k is scanned (in Line 14), contradicting the assumption that it was skipped. ■

Notice that all patterns that are fully contained within a single COPY instruction are found using the *Matched* list in Lines 21–25. Hence, Lemmas 1 and 2 along with this fact, immediately yield the following theorem:

Theorem 3 *Our algorithm reports exactly the same matched patterns as Aho-Corasick algorithm, which runs on the plain-text.*

C. Optimizations

In this section, we will present several optimizations to our basic algorithm to enhance its online running time. These optimizations trade running time with modest memory increase.

1) Efficient pattern lookups in the Matched list: Let n be the length of the *Matched* list. The pattern lookups in the *Matched* list is performed in Lines 21–25 of Algorithm 2 and runs in $O(n)$. A common way to reduce this running time is to save the *Matched* list as a balanced tree or as a *skip list* [22]. In such a case, the running time to find the first index which is larger than the copied address is only $O(\log n)$; then, the elements of *Matched* are checked one by one until the first element with index outsize the copied data is encountered. Another option, that trades memory with time, is to add an array of pointers, denoted by *MatchedPointers*, of the dictionary size (that is, with the same size as the *State* array). Element $MatchedPointers[i]$ contains a pointer to an element $\langle x, P\rangle \in Matched$ such that x is the smallest index that is greater or equal to i. This data structure reduces the running time to find the first index which is larger than the copied address to $O(1)$ (a single lookup in the *MatchedPointers* array).

Alternatively, given a COPY (x,p) instruction, one can cache the corresponding internal matches within $[p, p+x-1]$ in a hash-table whose key is "(x,p)". In such a case, when the exact COPY appears again, one can obtain all the matches in a single access to that hash-table. Since the dictionary usually contains common phrases (e.g., HTML commands) that are used again and again, our experiments show that such a cache is extremely efficient. Specifically, during a search on 100 delta files with the same dictionary, the portion of cache hits rises very fast and becomes almost 100%. After a sharp learning curve, the average cache hits on the last 90 files was 99.4% and the average on the last 50 files was 99.7%.

2) Eliminating the traversal of failures paths: Failure path traversal is done at the end of the COPY instruction processing in Lines 18–19 of Algorithm 2.

Notice that the failure paths depends only on the state saved by Algorithm 1, thus one can perform the failure path traversal in the offline phase and save the entire path in the *State* array. For example, $State[8]$ of the example of Section III-A, which contains the state s_{10} can be replaced by the entire failure path starting at s_{10}: (s_{10}, s_5, s_7, s_0). Furthermore, this path can be saved in a tree-based structure according to the depth of the different states. Thus, in order to find the appropriate state in Line 20, one should only perform a logarithmic (in the failure path length) number of operations.

In practice, failure path traversals impose almost no overhead on the running time of Algorithm 2 (see Section V).

D. Dealing with Gzip over SDCH

SDCH delta file are usually compressed by Gzip to decrease its size even further. In such a case, the client needs to first decompress (using Gzip) the file and then to decompress it using the dictionary to get the plain-text. Our algorithm requires that the compressed traffic has to be unzipped first; then, instead of decompress the SDCH compression, the resulting delta file is passed to our algorithm.

We note that Gzip decompression is a cheap operation compared to pattern matching. Furthermore, as mentioned above, SDCH compression ratio is around 44%, i.e. 56% of the data is copied from the dictionary. This numbers imply that even if the algorithm has to unzip the delta file before it scans it, we still improve the performance by around 56%.

IV. Regular Expressions Inspection

Regular expressions become an integral part of patterns that are used for security purposes. In Snort, an open source network intrusion prevention and detection system [23], 55% of the rules contain regular expression. Each regular expression pattern contains one or more string tokens along with one or more regular expression tokens. For example the regular expression `\d{6}ABCDE\s+123456\d*XYZ$` has the string tokens `ABCDE`, `123456`, `XYZ`, and the regular expression tokens `\d{6}`, `\s+` and `\d*`.

Like Snort, we treat the string tokens as *anchors* and insert them to the DFA. Only when all the anchors of a single regular expression pattern are matched, the regular expression tokens are examined (e.g., using a regular expressions engine). Furthermore, in most cases, we can limit the pattern search in at least one direction; namely, if before the first (resp., after the last) anchor, all tokens have a limited size (i.e., do not contain '+' or '*'), there is a bounded number of characters we should examine before (resp., after) the matched position of the anchor. In the above example, if we matched the anchor `ABCDE` at position x_1 and the anchor `XYZ` at position x_2, the left bound, l_{bound} is $x_1 - 10$ and the right bound, r_{bound}, is x_2. Thus, the interval $[x_1 - 10, x_2]$ should be passed to the regular expressions engine for re-examination. Note that l_{bound} can be 0 and r_{bound} can be the size of the file if there is an unlimited length token before the first anchor or after the last anchor.

To conclude, our regular expression inspection works as follows: First, constant strings (a.k.a anchors) are extracted from the regular expression offline. Then, our algorithm is applied on the SDCH-compressed traffic with the anchors as the patterns set. The anchors have to be matched in the same order of there appearances in the pattern. We save all the possible l_{bound} values (derived by the matched positions of the first anchor) and the maximal value of r_{bound}. Finally, we check if there is a regular expression which all its anchors were matched. If there is, for each l_{bound} value, we run an off-the-shelf regular expression engine from this value until, either we scan a character that yields a mismatch, or we have a full pattern match. In most of the cases, as we detail in Section V, we are scanning a few bytes for each l_{bound} value, and the total number of scanned bytes in the interval is significantly less than its size. Note that since we match the anchors at the correct order, the last scan is guaranteed to end before r_{bound}.

V. Experimental Results

Data Sets: Nowadays, Google's servers are the most prominent web servers that use shared dictionary compressed traffic. Furthermore, Google search is a fertile ground for popups, banners or any objectionable content. Thus, we evaluate our algorithm with Google search result files. We first downloaded the dictionary from `google.com` and used the 1000 most popular Google search queries (for each such query, we constructed an HTTP request and got a SDCH-compressed webpage, which we use as an input file).

The signatures data sets are drawn from a snapshot of Snort rules as of October 2010 [23]. We note that shared dictionary compressed traffic is mainly used on the traffic from the server side to the client side, which is not the case supported by most of the underlying rules within Snort. Still, in order to perform experiments using regular expressions we have extracted all Perl Compatible Regular Expressions (PCREs) from rules matching two header groups differing in the destination port (`any` and `$HTTP_PORTS`). There are 40 rules from the former group and 423 rules from the latter.

Since the input files do not contain many matches, specifically long matches are rare, we also constructed for each input file a *synthetic patterns file*, in the following manner: We calculated the length distribution of Snort's patterns and we randomly pick lengths from this distribution. For each length value ℓ, we took a sequence of ℓ characters from the uncompressed version of the input file and add them as a pattern to the patterns set for the specific input file. We stop when the total length of the selected patterns is equal to the input file size. As a result, we have 1000 input files, each with its own patterns file, such that each pattern in that file has at least one match in the corresponding input file and all the bytes in each input file belong to at least one pattern. We call this file *the synthetic case with 100% matches*. We also created *synthetic patterns file with 50% matches*, by randomly

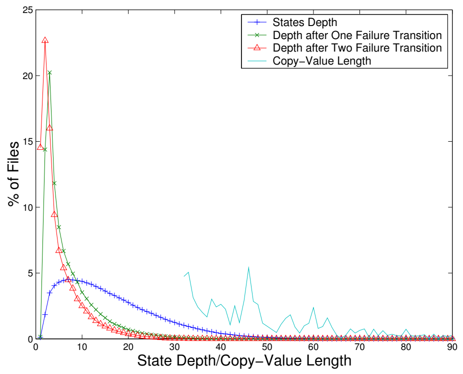

Fig. 2. The depth of first three states of each failure path (synthetic case with 100% matches) compared to the length of the COPY instructions.

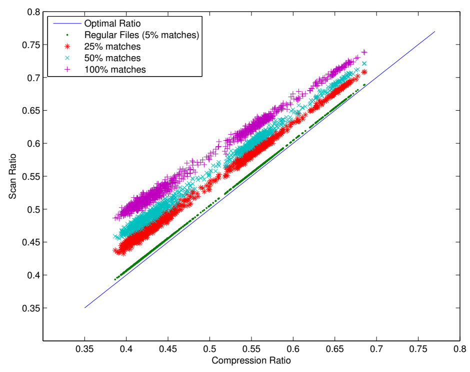

Fig. 3. Comparison between the *scan-ratio* and *compression-ratio* for the real-life and synthetic cases. The line depicts the best achievable *scan-ratio*.

selecting half of the patterns of the previous pattern file. Similarly, we also created a *synthetic patterns file with 25% matches*.

Time Reduction: We compared the execution time of our algorithm to an execution time of a naïve algorithm that first decompresses the file and then applies the Aho-Corasick algorithm on the plain-text. Our experiments show a significant improvement: on average, the execution time of our algorithm is only 34% of the time it takes the naïve algorithm to complete. It is important to note that some components of the naïve implementation are done using off-the-shelf software. Hence, we chose to take the following conservative approach in estimating the time reduction; our approach ignores completely the decompression stage of the naïve implementation and therefore it underestimates our performance gain.

Notice that our algorithm run time depends on the number of scanned bytes (Algorithm 2, Lines 6, 9, and 14) and the failure transitions it takes (Algorithm 2 , Lines 18–19). Thus, our main figure of interest is the ratio between the number of the bytes our algorithm scans and these failure transitions, in addition to the size of the plain text (scanned by the naïve algorithm); namely,

$$\text{scan-ratio} = \frac{\text{scanned bytes} + \text{failure transitions taken}}{\text{size of the plain text}} \quad (1)$$

Note that the different bytes of the plain-text can be classified by the type of their corresponding SDCH instruction. Let $|add|$ be the number of bytes generated by either ADD or RUN instruction (the number of bytes generated by a RUN instruction is negligible) and $|copy|$ be the number of bytes generated by a COPY instruction. Furthermore, we denote by $|copy_{scanned}|$ the number of bytes generated by a COPY instruction and scanned by our algorithm (that is, in Line 14). Thus, Equation (1) can be rewritten as

$$\text{scan-ratio} = \frac{|add| + |copy_{scanned}| + \text{failure transitions taken}}{|add| + |copy|}$$

Our experiments show that even in the synthetic case with 100% matches, in most of the files the algorithm takes no more than 2 failure transitions. The maximum number of failure transitions the algorithm takes is 19 (it occurs only in a single file), and the average number of transitions per file is 2.35; this is extremely low considering an average of 557 COPY instructions per file. This low number is explained by examining the relation between the length of the COPY instructions and the depth of the states in the *State* array (corresponding to the dictionary file), as shown in Fig. 2. In any case where that length is larger than the depth of the state, no failure transition is taken (recall Line 18 in Algorithm 2). We also depicted the histograms of depth after a single failure transition and after two failure transitions; these histograms show that even in the rare case where a failure transition is taken, the failure path traversal is stopped almost immediately.

We compare the *scan-ratio* with the *compression-ratio* $|add|/(|add|+|copy|)$; namely, the fraction of bytes that were generated using ADD instructions compared to the total number of bytes (or equivalently, the ratio between the size of the compressed text and the plain text).

Note that the best achievable *scan-ratio* is equal to the *compression-ratio* (when $|copy_{scanned}| = 0$). Furthermore, the better the *compression-ratio* is, the better the *scan-ratio* is, because more bytes are copied and therefore, potentially, less bytes should be scanned. Fig. 3 presents this exact relation for the input files with Snort‘s patterns as well as the synthetic cases. With Snort's patterns, the input files do not contain long patterns so the algorithm does not reach states with high depth. Therefore, the condition in Line 13 of Algorithm 2 rarely holds and the *scan-ratio* almost equals the *compression-ratio* for any *compression-ratio* value. In the synthetic cases, the input files contain patterns of all lengths and the algorithm reaches states with high depth. In these cases, the algorithm has to scan several bytes until the number of scanned bytes is equal or greater than the depth of the current state. Thus, for the synthetic cases the *scan-ratio* is between 1.03 to 1.2 times the *compression-ratio*, and it depends both on the *compression-ratio* value and the match percentage.

To conclude, these figures imply that we achieve almost optimal time reduction, which equals to the compression ratio.

Regular Expressions: The scan ratio with regular expression patterns was evaluated by calculating the number of extra bytes the algorithm scans when it matches all the anchors of such a pattern. Since matches are in general infrequent [24],

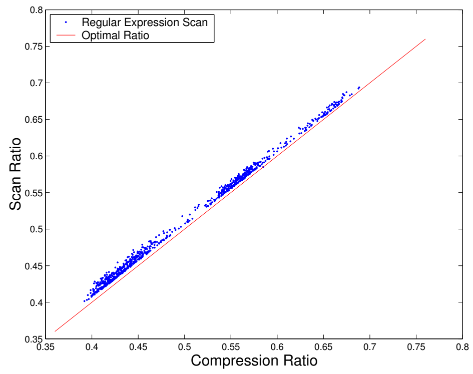

Fig. 4. Comparison between the *scan-ratio* when considering also regular expression matching and *compression-ratio* for the real-life case.

the regular expression engine is not executed often (on average, we have only 5 executions per file). Furthermore, even when all the anchors are matched, the engine scans only a few bytes, whose number is denoted by $regexp_{scan}$. The scan ratio from Equation V is redefined to include these characters by adding $regexp_{scan}$ to its numerator. Fig. 4 presents the relation between the compression ratio to the redefined scan ratio. On average, the overhead of the regular expression is around 1%, and the scan-ratio stays remarkably close to the optimal value. Note that the minimal scan ratio of any regular expression engine that scans the uncompressed file is 100%, i.e. every byte is scanned at least once.

Memory Consumption: Let n be the size of the dictionary, k be the size of the $Matched$ list, and p the number of bits required to represent a pointer in the system. The memory consumption of our algorithm is $np+2kp$, where the first term is for holding the values of the elements in $State$ array, and the second term is for the $Matched$ list (each of its k element holds a pointer to the dictionary and a pointer to the patterns).

Searching for Snort patterns on `google.com` dictionary yields a $Matched$ list of size approximately 40000 (we ignored matches of patterns of length 1 which can be dealt separately). In our given input, we need no more than 17 bits to address either a byte in a dictionary or a state in the DFA, therefore the memory consumption is 3457000 bits $=$ 420 KB for $n =$ 116 KB. This memory consumption can be further reduced using a variable-length pointer encoding.

VI. Conclusions

This paper presents a novel pattern matching algorithm on shared dictionary compressed traffic, which scans 99% of the bytes only once: around 56% of these bytes are scanned in an offline phase, implying that it gain up to 56% improvement in the performance over multi-patterns matching algorithm that scans the plain-text. In the worst case, our algorithm scans no more than the number of bytes in the uncompressed file. In addition, our algorithm has low memory consumption (around 420 KB for today's dictionary).

Our algorithm can run on two different environments. First, it can run within a security tool that performs DPI and therefore has to be deployed with a pattern matching algorithm. In addition, it can run in a single user environment, such as PC, tablet or cellular phone. The performance of all these tools is dominated by the speed of their string-matching algorithms [25], therefore our algorithm provides a real improvement when dealing with SDCH-compressed traffic. In addition, due to its low memory footprint, the algorithm can be easily deployed in nowadays environments.

Our future work includes methods to exploits the interplay between a Gzip compression which is performed over an SDCH-compressed file.

References

[1] W3Techs, "Usage of compression broken down by ranking." [Online]. Available: http://w3techs.com/technologies/breakdown/ce-compression/ranking

[2] P. Deutsch, "Rfc1952 gzip file format specification version 4.3," IETF RFC 1952, 1996.

[3] ——, "Deflate compressed data format specification version 1.3," IETF RFC 1951, 1996.

[4] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, pp. 337–343, May 1977.

[5] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of IRE*, pp. 1098–1101, 1952.

[6] B. M. J. Butler, W. Lee and K. Mixter, "A proposal for shared dictionary compression over http," Google Inc., Tech. Rep., 09 2008.

[7] SureStart, "Google chrome tops 20% market share for june 2011." [Online]. Available: http://www.sure-start.com/google-chrome-tops-20-market-share-for-june-2011/3675556/

[8] T. Virki and S. Carew, "Google topples nokia from smartphones top spot," Reuters. [Online]. Available: http://www.reuters.com/article/2011/01/31/us-google-nokia-idUSTRE70U1VW20110131

[9] Canalys, "Googles android becomes the worlds leading smart phone platform." [Online]. Available: http://www.canalys.com/pr/2011/r2011013.html

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

[11] B. W. Watson and G. Zwaan, "A taxonomy of keyword pattern matching algorithms," Eindhoven University of Technology, Tech. Rep. 27, 1992.

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

[13] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Department of Computer Science, University of Arizona, Arizona, Tech. Rep. TR-94-17, May 1993.

[14] A. Amir, G. Benson, and M. Farach, "Let sleeping files lie: Pattern matching in z-compressed files," *J. Comput. Syst. Sci.*, pp. 299–307, 1996.

[15] T. Kida, M. Takeda, A. Shinohara, and S. Arikawa, "Shift-and approach to pattern matching in lzw compressed text," in *CPM*, 1999, pp. 1–13.

[16] G. Navarro and M. Raffinot, "A general practical approach to pattern matching over ziv-lempel compressed text," in *CPM*, 1999, pp. 14–36.

[17] G. Navarro and J. Tarhio, "Boyer-moore string matching over ziv-lempel compressed text," in *CPM*, 2000, pp. 166–180.

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

[19] U. Manber, "A text compression scheme that allows fast searching directly in the compressed file," *ACM Trans. Inf. Syst.*, pp. 124–136, 1997.

[20] N. Ziviani, E. S. de Moura, G. Navarro, and R. Baeza-Yates, "Compression: A key for next-generation text retrieval systems," *Computer*, vol. 33, pp. 37–44, November 2000.

[21] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *IEEE INFOCOM*, may 2007, pp. 1064 –1072.

[22] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. of the ACM*, no. 6, pp. 668–676, 1990.

[23] "Snort," http://www.snort.org.

[24] A. Luchaup, R. Smith, C. Estan, and S. Jha, "Speculative parallel pattern matching," *will appear in June 2011 issue of the IEEE Transactions on Information Forensics and Security*, 2011.

[25] 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.