Accelerating Regular Expression Matching Over Compressed HTTP

Abstract— **This paper focuses on regular expression matching over compressed traffic. The need for such matching arises from two independent trends. First, the volume and share of compressed HTTP traffic is constantly increasing, mostly with GZIP that uses back-references to repeated strings. Second, due to their superior expressibility, current Deep Packet Inspection engines use regular expressions more and more frequently.**

We present an algorithmic framework to accelerate such matchings, taking advantage of information gathered when the traffic was initially compressed. Our algorithm is based on calculating (for every byte) the minimum number of (previous) bytes that can be part of a future regular expression matching. When inspecting a back-reference, only these bytes should be taken into account, thus enabling one to skip repeated strings almost entirely without missing a match. We show that our generic framework works with either NFA-based or DFA-based implementation and gains performance boost of more than 70%. Moreover, it can be readily adapted to most existing regular expression matching algorithms, which usually are based either on NFA, DFA or combinations of the two. Finally, we discuss other applications in which calculating the number of relevant bytes becomes handy, even when the traffic is not compressed.

I. Introduction

Deep packet inspection (DPI) is a crucial component in many of today's networking applications, such as security, traffic shaping, and content filtering. It is considered a system performance bottleneck, since it involves inspecting the packet's payload in addition to its header. Recently, especially due to the proliferation of mobile devices with limited bandwidth, DPI components have to deal also with compressed traffic. This adds an additional performance penalty as the data have to be decompressed prior to being inspected. Finding an efficient solution is crucial as nowadays *the majority of Web-sites uses compression*.[1] HTTP compression is done with the DEFLATE/GZIP protocol, which uses pointers to repeated strings within the traffic. Current literature focuses on DPI over compressed traffic for patterns that consist of strings rather than regular expressions. However, contemporary DPI engines are required to support also regular expressions.

This paper aims at providing a *generic* solution to accelerate *any* regular expression matching on compressed traffic. As such, it applies to a wide range of methods for regular expression matching over plain-text (namely, uncompressed) traffic. The inspection is accelerated by avoiding scanning repeated strings within the input text (namely, the strings represented as pointers in DEFLATE/GZIP), which were in a sense "already scanned". Extra care should be taken to detect and handle delicate cases such as those where the regular expression matches consecutive repeated and non-repeated strings (e.g., a pattern prefix followed by a pointer to a repeated string). DPI algorithms rely mostly on finite automata. In case of string matching, every state within the automaton corresponds to a single string. Therefore, storing the information about the previously traversed states is sufficient to determine the amount of bytes of the input text that may be safely skipped when encountering a pointer within a compressed input (as discussed in Section II). In case of regular expressions, every state may represent a wide set of strings with various lengths, as for example, in the presence of the `'*'` operator. For instance, given the pattern `'ab+c'`, the input strings `'abc'` and `'abbbbc'`, cannot be distinguished based on the information of the automaton state alone as both input strings scan leads to the same state. To overcome this problem, we provide an algorithm that evaluates a new parameter called *Input-Depth*, which relates to the length of the input shortest suffix that leads to the current state from the automaton's root.

We provide an algorithmic framework called *Acceleration of Regular expression matching over Compressed HTTP* (ARCH), which uses the *Input-Depth* parameter. Two system designs are derived from this framework with respect to the two distinct DPI approaches, namely: **two phase inspection** — string based pre-filtering accompanied by an non-deterministic finite automaton (NFA) scan for regular expression matching, and a **single pass inspection** — deterministic finite automaton (DFA)-based regular expression matching. Our experiments show that the first system design, denoted by ARCH-NFA, skips up to 79% of the inspected traffic and thus gains more than 4.8 times performance boost with respect to the second phase of regular expression matching. This design has a significant practical importance as it relates to the architecture used by the popular Snort [2] intrusion prevention system (IPS). Our second system design, denoted by ARCH-DFA, also skips up to 79% of the inspected traffic and gains more than 3.4 times performance boost for moderate size pattern sets. Next we show how ARCH applies on large pattern sets that require a multi-DFA design to avoid state-space explosion. This design maintains almost the same average skip ratio of 78% and gain performance boost of 3.3 times.

This paper has the following contributions:

1) Study the challenges of regular expression matching over compressed traffic.
2) A new method to allow the extraction of entire matched sub-strings for automata-based matching.
3) The first algorithmic framework that accelerates regular expression matching over compressed traffic.
4) Two architectural designs that achieve a significant performance improvement over traditional regular expression matching.

[1] A recent report from July 2014 shows that over 57.8% of the Internet Web sites use HTTP compression. When looking at the top 1000 most popular sites, over 83% of them use HTTP compression [1].

II. Background

A. Compressed HTTP

Compressed HTTP (namely, *content coding* for HTTP) is a standard method defined in RFC 2616, which standardizes HTTP 1.1 [3]. Three content coding methods are described in that RFC: GZIP, DEFLATE and COMPRESS. Practically, only the former two coding schemes are used. Since GZIP is a variant of DEFLATE, this paper handles both algorithms in the same way. The GZIP algorithm [4] has two underlying compression techniques: LZ77 and Huffman coding.

1) **LZ77 compression** reduces data size by identifying repeated strings within the last 32KB of uncompressed data. The algorithm replaces each such repeated string by a pointer with (distance, length) format, where *distance* indicates the distance in bytes of the repeated string from the current location and *length* indicates the number of bytes to copy from that point. For example the string: '`abcdefabcx`' may be compressed to '`abcdef(6,3)x`'.
2) **Huffman coding** further reduces the size of the data by assigning variable-size code-words for *symbols*, thus enabling to encode frequent symbols with fewer bits.

GZIP compression first compresses the data using LZ77 compression and then it encodes the literals and pointers using Huffman coding. Specifically, the algorithm in this paper is based on the characteristics of the LZ77 compression.

B. Deep Packet Inspection Algorithms

DPI involves processing of the packet payload to identify occurrences of a set of predefined patterns. These patterns are expressed as either a string or a regular expression. Early network IPSs relied solely on string matching, which is usually based on some variant of the Aho-Corasick [5] algorithm. This method uses a DFA to recognize multiple string signatures in a single pass and its running time is deterministic and linear in input size.

Over the last years, security threats become more sophisticated and require complex signatures, which can no longer be expressed as strings. Therefore, most security tools incorporate a regular expression matching engine. Regular expressions add basically three operators to string matching: concatenation (of two expressions), alternation (OR, '`|`') and Kleene closure ('`*`' or '`+`') and are defined recursively over these operators [6]. Such engines are typically implemented using either a DFA or an NFA.

DFA scans the input in linear time, but uses huge amount of memory due to the infamous state-space explosion problem. Approaches to tackle this problem include keeping cache of recently visited states [7], adding instructions to automaton edges [8], using edge compression techniques [9], [10], and using multiple DFAs [11] where only some are activated [12]. Our ARCH-DFA solution is based on A-DFA [10] and the design that handles large pattern sets is inspired by Hybrid-FA [12].

NFA, on the other hand, uses linear space but has worst-case quadratic time complexity [6], which may be exploited by an adversary for denial-of-service attack (DoS) on the DPI element itself [13], [14]. In NFA, upon inspecting a symbol b in a current state s, one may need to transit to multiple next states (namely, by traversing all the outgoing edges of states s with label b) . A common implementation is to maintain at each time a set of *active states* S; for each input byte b this set should be recomputed by traversing all b-labeled edges from every state $s \in S$.

In practice, security tools such as Snort [2], which has a large rule-set, uses a pre-filter based solutions. The pre-filter performs string matching on extracted strings from the regular expressions. Upon a match of all strings of a specific rule, the full regular expression scan is invoked using an NFA. The pre-filter itself can be accelerated in the case of compressed traffic based on prior art that provide solutions for string matching (as described in the following subsection), while the NFA part may be accelerated using our ARCH-NFA solution.

C. String Matching Over Compressed Traffic

LZ77 is an adaptive compression since each symbol is determined dynamically by the data. Therefore, there is no way to perform DPI without decompressing the data first [15]. Since decompression is a fairly inexpensive process, the real challenge is to *optimize the scanning process of the decompressed data*. The main idea behind the acceleration algorithm is to take advantage of information gathered by the decompression phase to skip scanning significant parts of the data. The LZ77 pointer represents a repeated string, and more specifically, one that was already scanned. Therefore, it is possible to skip scanning most of the bytes represented by the pointer without missing any pattern. Still, a naïve approach of just keeping track of previous matches does not suffice due to a case where a pattern crosses a border of a repeated string. For example, given a pattern'`nbc`' and an input string '`abcdnbxn(7,5)c`', while no match occurred at the repeated string'`bcdnb`', there are matches of both its left and right borders.

This problem is tackled by the ACCH algorithm [15], *which works only on string matches and cannot handle regular expressions*. In a nutshell, ACCH applies a DFA-based scanning and, upon encountering a repeated string, it works as follows:

1) Scan the *left border* of the repeated string and update scan results.
2) Check whether the previous scan results of the repeated string contains matches.
3) Scan the *right border* of the repeated string and update scan results.
4) Update estimated scan results of skipped bytes within the repeated string.

In ACCH, scan results of previous bytes are stored in a 32K-entries *Status-Vector*. Status values are determined by the state s which is reached when scanning the byte. Specifically, it depends on the *DFA-Depth* of state s, which is defined as the length of the shortest path from root to s. There are three possible status values:

- MATCH: a pattern was matched (ends in the scanned byte).

- UNCHECK: *DFA-Depth(s)* is below a threshold parameter *CDepth* (in practice, *CDepth* is set to 2).
- CHECK: otherwise.

ACCH uses the *Status-Vector* in the following manner:

Left Border Scan — Upon processing a repeated string, ACCH scans j bytes in the repeated string until reaching a state s where $j \geq$ *DFA-Depth(s)*. From this point on, any pattern prefix is already included in the repeated string area. We refer to the first j bytes of the repeated string as the *left border* of the pointer.

Repeated Pattern Detection — This procedure examines the values of the bytes of repeated string in the *Status-Vector*, starting at its $(j+1)^{\text{th}}$ byte. If some byte m has a MATCH value, ACCH estimates the *DFA-Depth* of a state reached by the DFA had the scan continued until m. If this estimation is m or more, then the previously matched pattern was not repeated entirely. Otherwise, ACCH reports a matched pattern according to the pattern within the repeated string and marks that byte by MATCH status. ACCH estimates the *DFA-Depth* value of a skipped byte using the status of the corresponding referred byte. Specifically, an UNCHECK status implies that the byte's *DFA-Depth* is at most *CDepth*. As a CHECK status does not bound *DFA-Depth*, ACCH searches for the last UNCHECK occurrence prior the position m of the referred byte. If UNCHECK was located n bytes prior to m, then the *DFA-Depth* can be bound by *CDepth*+n.

Right Border Detection — ACCH should determine the position from which it starts inspection again so that no pattern (whose prefix is part of the repeated string suffix) is missed. This is done by estimating the *DFA-Depth* of the repeated string's last byte (say this estimation is d) and scanning (from s_0) the last d bytes of the repeated string.

Update Status of Skipped Bytes — Skipped bytes cannot obtain their status from the scan procedure. Therefore, ACCH copies the status value for the skipped bytes from the repeated string. Note that since ACCH skips only after it ensures that there is no pattern whose prefix started prior to the repeated string, the retrieved value of the estimated *DFA-Depth* could be only equal or higher than its actual value (namely, setting a byte as CHECK while it should be UNCHECK). Notice that such a mistake leads to minor reduction in the amount of skipped bytes but never to a miss-detection.

III. Related Work

HTTP compression is a very popular method in the Internet. There are many works (e.g., [16]–[18]) that suggest acceleration of the compression method itself to support high request volume from high-end servers.These papers support the compression layer of the data without referring to the context of processing the data itself as in the case of DPI.

As noted in Section II, HTTP compression uses LZ77 compression. There are various researches in the literature regarding pattern matching methods over the Lempel-Ziv compression family [19]–[22]. However, the LZW/LZ78 variants are more attractive and simple for pattern matching than LZ77, thus all the above works are not applicable to our case. Klein and Shapira [23] suggested a modification to the LZ77 compression to simplify matching in files. However, their suggestion is not implemented in today's Web traffic. Farach et. al [24] deals with pattern matching over LZ77. However, the algorithm matches only a single pattern and requires two passes over the compressed text (file), which does not comply to the 'on-the-fly' processing requirement of network domains.

The ACCH algorithm [15], discussed in details in Section II, was the first algorithm to tackle the problem of multi-pattern matching over compressed HTTP traffic. Following this work, another algorithm, named SPC [25], analyzes the usage of the Wu-Manber pattern matching algorithm [26] instead of the DFA-based Aho-Corasick [5] algorithm that lies at the core of ACCH. SPC provides superior performance results over ACCH in the case of normal traffic while its worst-case performance is very poor. Note that there is no variant of the Wu-Manber algorithm that is applicable for regular expression matching, thus SPC cannot be used in our case. Another algorithm, named SOP [27], focuses on minimizing the memory footprint required by the ACCH data structures. SOP algorithm, with some few adaptations, may be combined with our algorithm to save space. Finally, Berger and Mortensen [28] provide a hardware scheme that supports the decompression phase, but does not deal with the scanning of the compressed data itself.

All the above mentioned works are limited to *exact string matching* systems rather than to devices based on regular-expression–matching. Sun et al. [29] suggested a method of performing regular expression matching over compressed HTTP. Still that method handles only simple cases, where the DFA is either at its root state or at a state with a direct transition from the root state. Practically, the DPI engine may traverse to deeper DFA states. Furthermore, an attacker may easily craft an input that causes the DFA traversal into areas in which the algorithm of Sun et al. fails to support. Robustness against such attacks is crucial as IDS's are a preferred target for denial-of-service attacks.

IV. The ARCH Framework

This section presents our framework: *Acceleration of Regular expression matching over Compressed HTTP* (ARCH). Conceptually, ARCH is based on the same ideas as the ACCH algorithm (as described at Section II-C), which works only for string matchings. One of the key insights used in ACCH is that the *DFA-Depth* of a state represents the longest suffix of the input that can still be part of a (future) match. This property holds for string matches and it greatly simplifies the design of ACCH, as it enables both left- and right-border resolution based solely on the state of the DFA.

Unfortunately, for regular expression matching this property does not hold since a state may represent an input of variable lengths. This can happen as a result of an alternation between different-length expressions or in the presence of a Kleene closure (namely, `'*'` or `'+'`). An example of a state with an ambiguous depth in the presence of an alternation operator is depicted at Fig. 1, which represents an automaton for the pattern `'(apple|pear)s'`. The *DFA-Depth(s_5)* value is 4 since it is its shortest path length from s_0. Therefore, in the case of an input byte sequence of: `'zpplesxa(7,5)s'`, the *DFA-Depth* after scanning the string `'apple'` at the repeated

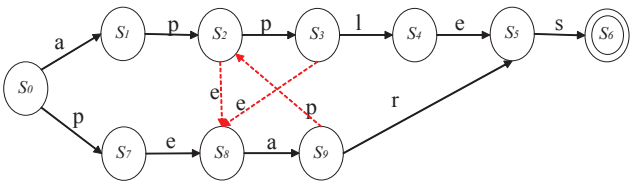


Fig. 1. DFA for `(apple|pear)s`. Failure transitions are marked with red dashed arrows. Failure transitions to states with *DFA-Depth*s 0 and 1 are omitted for clarity.

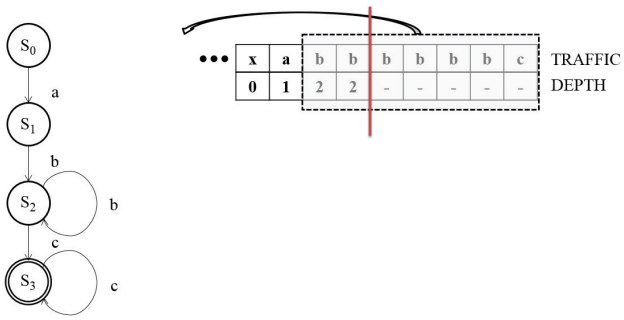


Fig. 2. DFA for `ab+c+`. The right table indicates the input data and its *DFA-Depth* as derived from the automaton while scanning the input.

string would be 4, while the minimal input suffix that should have been scanned to reach s_5 from root is 5. Wrong *DFA-Depth* may lead to miss-detection in ACCH.

An example of a state with an ambiguous depth in the presence of a Kleene closure, is demonstrated in Fig. 2. The DFA is for `ab+c+` and an input of `bbbbbbcxa{8,7}` for which the repeated string is `bbbbbbc`. The repeated string scan starts at s_1. After scanning two `b` characters the automaton is still at s_2, hence the DFA-Depth(s_2) does not change and is still 2. Thus, the left border detection of ACCH would have been completed at this point, skipping the rest of the repeated string and resuming scan at the right border. This, of course, leads to a miss-detection, which is unacceptable.

Therefore, a key challenge behind ARCH is to calculate the minimum number of (previous) bytes that can be part of a future regular expression matching. This number is captured by the *Input-Depth* parameter, which is defined precisely below. It is important to notice that unlike ACCH's *DFA-Depth*, *Input-Depth* depends both on the automaton state and the inspected input.

Definition 1. *For a given automaton, let s_0 be the start state and s be the current state after scanning input text T with the automaton. Let* Input-Depth(T, s) *be the length of the shortest suffix of T in which inspection starting at s_0 ends at s.*

We note that in the case of string matching, Input-Depth equals DFA-Depth. In fact, ARCH framework uses the ACCH algorithm to calculate possible scan skipping by replacing the DFA-Depth parameter with Input-Depth parameter along with specific implementations for the different setups, as described in the sequel. Thus, if there is no match, the *Status-Vector* value CHECK is determined according to whether *Input-Depth* is greater than $CDepth$ and if not the value is UNCHECK. The calculation of *Input-Depth* is done differently in NFA-based and DFA-based implementations, where in the former the value can be calculated precisely, while in the latter it can only be estimated. In the next sections, we will discuss these calculations and estimations as well as the correctness of the ARCH algorithm in detail.

TABLE I. SIMULATION OF ARCH OVER ACTIVE STATES NFA.

Input Character	Active States	Input-Depth Set	Max Input-Depth
z	s_0, s_1, s_7	$0, 1, 1$	1
f	s_0, s_7, s_8	$0, 2, 2$	2
\n	s_0	0	0
z	s_0, s_1, s_7	$0, 1, 1$	1
a	s_0, s_2, s_7	$0, 2, 2$	2
b	s_0, s_3, s_7	$0, 3, 3$	3
c	s_0, s_4, s_7	$0, 4, 4$	4
f	s_0, s_5, s_7, s_8	$0, 5, 5, 5$	5
e	s_0, s_7, s_9	$0, 6, 6$	6

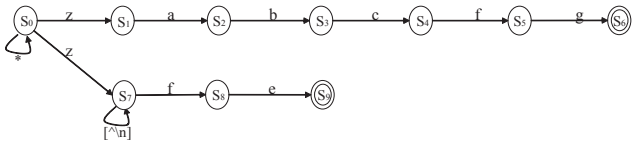


Fig. 3. NFA of pattern set: `zabcfg`, `z[^\n]*fg`.

V. INPUT-DEPTH CALCULATION FOR NFA-BASED IMPLEMENTATIONS

Recall that one possibility to provide regular expression matching is to use NFA, which is usually a compact but slow data structure. We start our discussion with such implementations as they are simpler and help to grasp a better understanding of ARCH (DFA-based Implementations pose extra complications on top of ARCH and are discussed in Section VI). Moreover, NFA is currently used by the popular Snort IDS, and therefore, accelerating its operations has a merit on its own.

ARCH maintains an *Input-Depth* value for each active state. Namely, given an input prefix T, an active state s, an input byte b, and a subsequent active state s' such that there is a transition between s and s' with a label b, the value of *Input-Depth(Tb,s')* is set to *Input-Depth(T,s)*+1. The only exception is at the start state of the NFA (which is always active) and for which *Input-Depth(T,s_0)* is always zero. Naturally, when ARCH needs to determine left or right border of a pointer it uses the $\max_{s \text{is active}} Input - Depth(T, s)$. Fig. 3 depicts an NFA for the patterns '`zabcfg`' and '`z[^\n]*fg`'. The execution of ARCH for input '`zf\nzabcfe`' and the NFA are illustrated in Table I.

VI. INPUT-DEPTH ESTIMATION FOR DFA-BASED IMPLEMENTATIONS

The task of *Input-Depth* calculation is more challenging when a DFA is used for regular expression matching. This is because, unlike NFA, a DFA transition may results either in increasing the *Input-Depth* (by one) or decreasing the *Input-Depth* (by any value). In this section, we provide an *upper bound* on the *Input-Depth* using two methods: by *simple and*

complex states and by *positive and negative transitions*. As a rule, we use the upper bound as the value of the *Input-Depth* in the ARCH framework. Thus, we take a conservative approach and never miss a match. Yet, the tighter the upper bound is, the higher the skip ratio we achieve.

A. Estimation based on Simple and Complex States

As noted above, *Input-Depth* cannot be always derived from *DFA-Depth*. Still, there are cases were we can derive it safely. In fact, we could split the DFA states into two kinds: those which represent a fixed string expression (where *Input-Depth* equals *DFA-Depth*) and those which represent a set of strings with various lengths. For instance, at the DFA of Fig. 2, states s_0 and s_1 are simple and states s_2 and s_3 are complex. More formally, this is captured in the following definition:

Definition 2. *A* simple *state s is a state for which all possible input strings that upon scan from s_0 terminate at s has the same length. All other states are* complex.

Input-Depth estimation by simple and complex states works as follows: upon traversal to a simple state, *Input-Depth* is set to the *DFA-Depth* of the state; upon traversal to a complex state, *Input-Depth* is incremented by one.

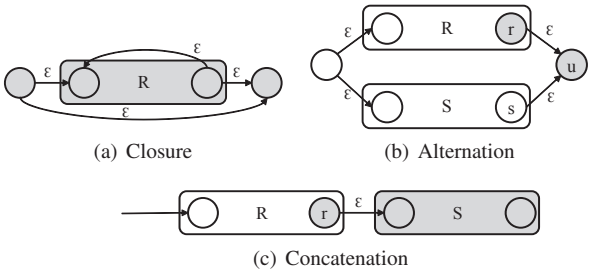
Our algorithm detects simple and complex states based on the DFA construction procedure as described by Thompson [30]. Thopmson's construction has three significant stages: NFA construction from expressions, ϵ-transitions removal, and finally, DFA construction based on the resulting NFA. The basic idea of our algorithm is that we mark simple and complex states during the NFA construction, and then we transfer these marks to the final DFA. The NFA construction is defined over the three basic regular expressions operators as in Fig. 4. The operands are regular expressions R and S. All states are marked as *simple* by default and stay that way unless specified otherwise. The complex states are marked according to each regular expression operator as follows:

- **Kleene Closure** (Fig. 4(a)): Mark all states as complex.
- **Alternation** (Fig. 4(b)): If either states r or s are complex, mark state u as complex.
- **Concatenation** (Fig. 4(c)): If state r is complex, mark all states of S as complex.

After the above procedure the following claim holds (the proof is at the appendix):

Claim 1. *If a state s is simple then all possible input strings, whose scan from s_0 terminates at s, have the same length.*

The simple/complex state marks are transferred through the Thompson construction stages to the final DFA as follows: the ϵ-transition removal stage only removes redundant transitions and states, therefore, all remaining states are still marked. Next, the NFA is transformed into a DFA using the *subset-construction method*, which traverses the entire NFA and creates a DFA state for each possible set of active states. This DFA state is marked simple if and only all the NFA states in its active states set are simple.



(a) Closure (b) Alternation

(c) Concatenation

Fig. 4. The basic operators considered in Thompson's construction [30].

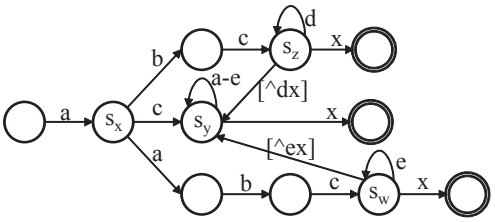


Fig. 5. DFA for pattern set {'`ac[a-e]*x`', '`abcd*x`', '`aabce*x`'}.

B. Estimation based on Positive and Negative Transitions

In this section we present the positive and negative transition technique which relies on the observation that *Input-Depth* depends on the *transition between two states* rather than only the state in its endpoint. For example, Fig. 5 shows that transition from s_x to s_y should increase the *Input-Depth*, transition between s_z to s_y should leave the *Input-Depth* unchanged, and transition between s_w to s_y should decrease the *Input-Depth* by 1. This implies that the *Input-Depth* calculation cannot solely depend on s_y.

Thus, we define two types of transitions: A *positive-transition*, which increases *Input-Depth* by one (e.g. a byte was added to the pattern's prefix) and *negative-transition*, which either decreases or leaves *Input-Depth* unchanged. The challenge is to detect negative-transitions and determine the exact change they apply on the *Input-Depth* value.

Negative-transition detection is performed in two stages. In the first stage, for each state we calculate a *candidate label*, denoted by $c\text{-}label(s)$, which helps detecting negative transitions in the second stage. Let $L(s)$ be the set of all input strings that are accepted by s. We choose $c\text{-}label(s) \in L(s)$ such that $c\text{-}label(s).length \leq l(s).length$ for any $l(s) \in L(s)$. We refer to the second stage of the Thompson construction (as described at Section VI-A), which is the subset-construction that constructs a DFA from an NFA. The $c\text{-}label(s)$ parameter is determined by integrating a line into the subset-construction algorithm (as in Alg. 1 Line 7). The algorithm uses the following basic NFA operations:

- ϵ-closure(T) — The set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone.
- move(T,a) — The set of NFA states to which there is a transition on input symbol a from some state s in T.

Basically, the subset-construction algorithm traverses all NFA active state sets (Line 1) using all possible input symbols

(Line 3) in a breadth-first search (BFS) manner. Each unique NFA active set T results in a corresponding DFA state (Line 6). When a transition from a DFA state T (which also represents an NFA active set) results in a DFA state U, which was never visited before, the algorithm creates a $c\text{-}label(U)$ by concatenating $c\text{-}label(T)$ and the transition label (Line 7). Note that there may be more transitions that lead to DFA state U along the construction. Still, since a BFS is used for the NFA traversal, and the $c\text{-}label(U)$ was generated upon the first transition that reaches U, $c\text{-}label(U)$ must be no longer than any other label in $L(U)$, as required.

```
Data: Dstates, Dtrans — containers for DFA states and
      transitions respectively. Initially, ϵ-closure({s0})
      is the only state in Dstates and it is unmarked.
1  while there is an unmarked state T in Dstates do
2      mark T;
3      foreach input symbol a do
4          U = ϵ-closure(move(T, a));
5          if U is not in Dstates then
6              add U as an unmarked state to Dstates;
7              set c-label(U)=c-label(T) ⊕ a;
8          end
9          Dtran[T, a] = U;
10     end
11 end
```

Algorithm 1: Modified subset construction algorithm.

At the second stage, for each state $T \in Dstates$ we iterate over all its transitions and determine whether they are negative or positive. For that we define a structure called *Anchored-NFA*, which is the NFA constructed from all regular expressions from the related DFA after they were *anchored*; namely, after they were restricted to be matched from the beginning of the input (represented by the `'^'` operator in the PCRE [31] package). Each transition (T, U) with label a is inspected as follows: the algorithm traverses the Anchored-NFA using $c\text{-}label(T)$ and receives an output of an active states set R in the Anchored-NFA (Line 2). If there exist an iteration with label a from R, then transition (T, U) is positive (Lines 4–6). The rest of the transitions are marked as negative. The intuition behind this algorithm is that a negative transition from T to U with label a applies that there is a suffix of $c\text{-}label(T) \oplus a$ that leads from the root to U. Therefore, a string such as $c\text{-}label(T) \oplus a$ cannot be accepted by the Anchored-NFA, rather only its suffix. Thus, lack of transition at the Anchored-NFA applies a negative transition at the DFA.

For example, consider the patterns '`ab+c+d`' and '`bc+e`'. The resulting DFA from our modified subset construction is depicted in Fig. 6(a) and its full-matrix representation along with its calculated c-label(s) values is depicted at Fig. 6(c). After running Alg. 2 using the DFA and the Anchored-NFA (depicted in Fig. 6(b)) all negative, positive (underline) and loop (circled) transitions are marked. All negative transitions are marked with dashed red arrows at Fig. 6(a). Note that the algorithm distinguishes between the "self-loop" of states {0}, {0,1} and {0,5} as a negative transition and the "self-loop" of states {0,2,5}, {0,3,6} and {0,6}, which are marked positive. This is the desired outcome as in the former case Input-Depth should not increase upon loop traversal, while in the latter it should.

```
1  foreach state T in Dstates do
2      R=Anchored-NFA(move(c-label(T)));
3      foreach input symbol a do
4          U=Dtran[T,a];
5          if ϵ-closure(move(R, a))≠ ∅ then
6              Dtran[T,a].positive=true;
7          else
8              Dtran[T,a].positive=false;
9          end
10     end
11 end
```

Algorithm 2: Marking positive/negative transitions.

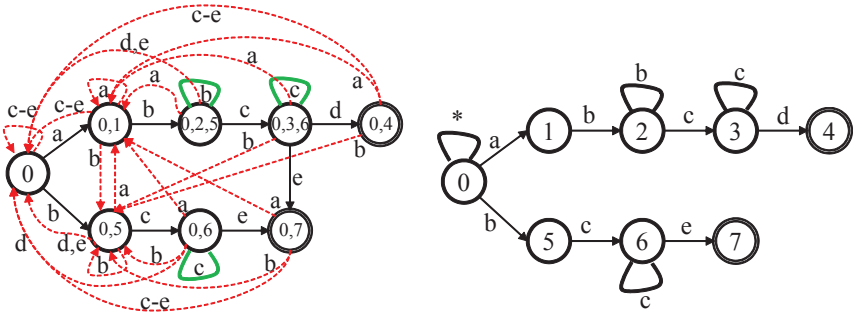
At this point, each transition at the DFA is marked as either positive or negative. The next step would be to determine the *Input-Depth* upon DFA traversal as described in Alg. 3. The straightforward case is upon a traversal over a positive transitions, for which the *Input-Depth* is incremented by one (Line 3). To handle negative transitions we use the classification of simple and complex DFA states as explained at the previous subsection. If a negative transition leads to a simple state, the Input-Depth is set to this state's DFA-Depth (Line 6). Upon a negative transition to a complex state the Input-Depth should be decremented by the delta between the labels' lengths as described at Lines 8–9. We note that this algorithm does not support some corner cases (e.g., when a regular expression contains multiple loops and has a negative transition to another regular expression that contains only some of these loops), thus our algorithm just increments *Input-Depth* by one as it is its the maximal possible value. For that matter ARCH always provides an upper bound on the value of *Input-Depth*, and therefore, never misses a match.

```
1  U=Dtran[T,a];
2  if Dtran[T,a] is positive then
3      Input-Depth++;
4  else
5      if U is simple then
6          Input-Depth=U.DFA − Depth;
7      else
8          Decrease=c-label(T) − c-label(U);
9          Input-Depth=Input-Depth − Decrease;
10     end
11 end
```

Algorithm 3: ARCH-DFA Input-Depth Maintenance.

VII. From Theory to Practice: ARCH-DFA System Architecture

Observe that the problematic case of ARCH is where the algorithm cannot skip any bytes. This case may happen upon a closure operator over a wide range of characters (such as '`.*`'), which causes its following states to be complex and have only positive transitions, hence the *Input-Depth* never decreases. For instance, consider the pattern '`AdminServlet.*(userid|adminurl)`' (which was extracted from Snort's rule set); after matching the prefix



(a) DFA (b) Anchored-NFA

c-label(s)		a	b	c	d	e
-	0	0,1	0,5	0	0	0
a	0,1	0,1	0,2,5	0	0	0
b	0,5	0,1	0,5	0,6	0	0
ab	0,2,5	0,1	0,2,5	0,3,6	0	0
bc	0,6	0,1	0,5	0,6	0	0,7
abc	0,3,6	0,1	0,5	0,3,6	0,4	0,7
bce	0,7	0,1	0,5	0	0	0
abcd	0,4	0,1	0,5	0	0	0

(c) Transition table

Fig. 6. Data structures used for the positive/negative transition marking for pattern set: {'`ab+c+d`', '`bc+e`'}.

'`AdminServlet`', *Input-Depth* cannot decrease since any character can be a part of the pattern and may be followed by the either '`userid`' or '`adminurl`'. Thus, ARCH cannot skip scanning characters in such cases as it never finds the *left border* of a pointer (see Section II-C). This implies that whenever a closure operator acts upon a wide range of characters, the effectiveness of ARCH reduces. In some cases, the benefit of ARCH becomes smaller than its overhead. In such cases we would like to just run a regular DFA rather than ARCH-DFA to avoid the ARCH-DFA overhead, which is approximately 11% (see details in Section VIII) due to additional memory accesses to the status-vector. We note that only 8.1% of the rules in the set that was derived from Snort and fits compressed Web traffic (i.e. Web traffic from server to client) contains '`.*`' expressions. A more frequent wide-range Kleene closure is of the form '`[^\n]*`'.Still, since we deal with HTML data, the new-line occurs on average after 75 characters, and therefore, the automaton "escapes" the above-mentioned case of only positive transitions, and therefore, does not harm much the ARCH performance, as described in Section VIII. Additionally, when *Input-Depth* increases along scanning, ARCH never detects a pointer's *left border*, implying status vector checkups are redundant. We note that nowadays, the incurred overhead in the above-mentioned case is practically negligible, future pattern sets or implementations might be different.

A design that ARCH may benefit from is the Hybrid-FA of Becchi et al. [12]. Inspired by this design, ARCH system breaks the regular expressions into a prefix set of all expressions, which are represented by a single *prefix ARCH-DFA* (which is analogues to the head-DFA in [12]), and a set of *tail ARCH-DFAs* which represent the entire regular expressions set. The *prefix ARCH-DFA* is always active, while only a small part of the *tail ARCH-DFAs* are also active. This way the resulting hybrid automaton has a small memory footprint as compared to the corresponding DFA, with the penalty of traversing several automatons in parallel. Using the above-mentioned architecture leaves us the freedom to decide in advance whether to use a *tail ARCH-DFA* or a *tail DFA* in the case where the overhead of ARCH is higher than its benefit as depicted at Fig. 7.

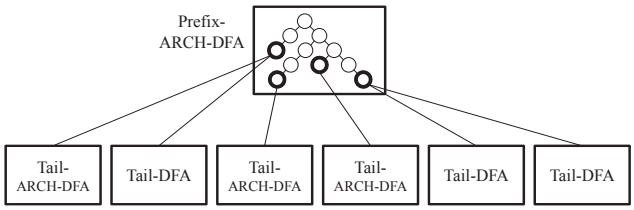


Fig. 7. Illustration of the Hybrid ARCH-DFA System architecture.

VIII. Experimental Results

In this section we evaluate the performance benefits of ARCH on rule-sets from the Snort IPS. The Snort24, Snort31, and Snort34 rule-sets were retrieved from Becchi's "Regular Expression Processor" package [32]. The Snort135 rule-set was extracted from Snort's *"web-client.rules"* (February 2014). These rules relate to Web traffic . Table II summarizes the basic characteristics of the rule-sets used in the experiments.

A. Data Set

The data set contains 2301 compressed HTML pages, downloaded by a Web crawler from 500 popular sites taken from the *Alexa* Web-site [33]. The size of the data is 358MB in uncompressed form and 61.2MB in compressed form.

B. ARCH performance

In this section we compare ARCH with a baseline algorithm, which uses the same underlying automaton scheme (NFA or DFA) without performing any byte skipping. We define R_s as the scanned character skip ratio — the ratio of characters scanned using ARCH out of the total size of the input data. We define R_t as the scan time ratio — ARCH's running time compared to the baseline algorithm's running time.

TABLE II. Rule-sets characteristics

Rule Set	Snort24	Snort31	Snort34	Snort135
Number of Rules	24	31	34	135
% Rules with Kleene Closure	37.5	41.9	38.2	89.6
% Rules with Char Ranges ≥ 50	50	48.4	41.1	51.9

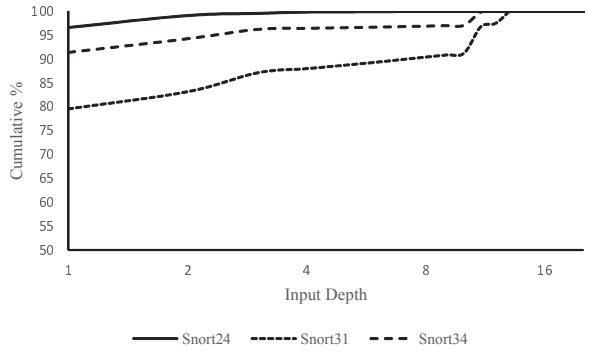


Fig. 8. Input Depth Cumulative Distribution (NFA)

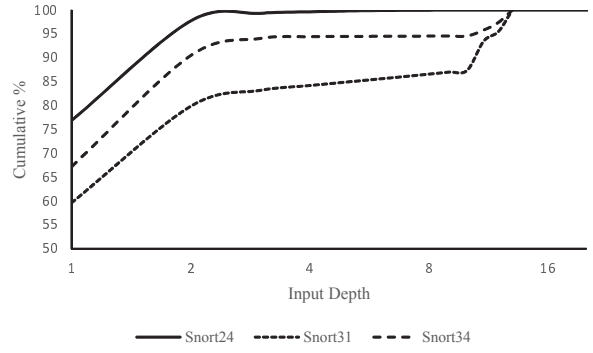


Fig. 9. Input Depth Cumulative Distribution (DFA)

ARCH-NFA was implemented using *active states NFA* as its baseline algorithm, as described in Section V. Table III shows statistics regarding its performance. The average skip rate R_s is 77.99%, which in turn results in a significant performance improvement in terms of R_t (77.21% on average). Compared to the overall performance, the overhead of ARCH-NFA is less than 1%. Since ARCH-NFA's running time is approximately 40 times longer than ARCH-DFA's, the relative overhead of ARCH is negligible. On the other hand, the space requirements for ARCH-NFA are 18 times smaller than those of ARCH-DFA.

ARCH-DFA was implemented using *A-DFA* [10] as its baseline algorithm. The implementation performs *InputDepth* estimation based on simple and complex states as described in Section VI-A. This estimation has a lower overhead while the benefit gained by the more precise estimation of the positive/negative transitions is not significant on our data sets. Two system setups were implemented for ARCH-DFA. The first uses a single *DFA* for the entire rule-set and is therefore useful for pattern sets with moderate size which do not exhibit state explosion, such as: Snort24, Snort31, and Snort34. Table III shows statistics regarding the performance of ARCH-DFA. ARCH-DFA achieved an average skip rate R_s of 77.69% and a performance boost in R_t of 69.19%. Note that in this case the overhead of ARCH is more notable and is almost 11%. As discussed in Section VII, this overhead is mostly due to additional memory references to the *Status-Vector* as compared to the baseline algorithm.

The second setup uses a hybrid design consisting of a prefix ARCH-DFA, which encodes the prefixes of the pattern set, and multiple tail ARCH-DFAs as described in Section VII. This setup is useful for pattern sets with considerable size and high complexity, which cause state-space explosion (e.g. Snort135). The average skip ratio R_s is 77.88% while the average gained performance boost in terms of R_t is 69.41%. We note that in this case, as in the simpler pattern sets, the overhead of ARCH is approximately 11%.

TABLE III. ARCH PERFORMANCE

	Rule Set	Snort24	Snort31	Snort34	Snort135
NFA	Characters Skipped	79.44%	75.86%	78.66%	-
	Time Saved	79.29%	74.56%	77.78%	-
DFA	Setup	Single	Single	Single	Hybrid
	Characters Skipped	79.04%	75.64%	78.38%	78.46%
	Time Saved	70.97%	67.46%	69.14%	70.08%
	Number of Matches	168603	1333	8000	77801
	Run-time NFA/DFA	38.46	45.45	40	-

IX. CONCLUSION

ARCH started as an effort to adapt prior art of string matching over compressed traffic to the regular expression matching domain in a straightforward manner. We uncovered wide set of complexities of both algorithmic and architectural aspects that the new domain holds. We aimed at providing a generic method that fits a wide range of solutions, therefore we analyzed both NFA and DFA setups and aimed at both moderate simple pattern sets and large complex ones. For that we proposed three architectures: ARCH-NFA, ARCH-DFA and the hybrid ARCH-DFA. These architectures were implemented and gained significant performance boost over current DPI designs of 67%–79%.

An important product of this research is the discovery of the *Input-Depth* parameter. Beside the fact that it is a crucial construct for regular expression matching over compressed traffic, we found that there are other applications for which it is important. A surprising fact is that there is no straightforward method to extract the string that relates to a matched pattern without rescanning the packet. The only information about the input that is available at this point is the position in the input in which the match occurred. Input-Depth allows this functionality as it indicates the number of bytes that should be extracted from the input up to the match position.

A related application may be to determine the number of bytes that should be stored to handle cross-packet DPI. Instead of buffering entire packets, a DPI engine may just store a matched pattern-prefix at each packet's suffix. This application has even more benefit in the case of out-of-order packets, where in order to be able to retrieve a matched string several packets should be buffered.

REFERENCES

[1] W3Tech, "Usage of Compression for websites," 2014. http://w3techs.com/technologies/details/ce-compression/all/all.

[2] "Snort: The Open Source Network Intrusion Detection System." http://www.snort.org.

[3] "Hyper[t]ext [t]ransfer [p]rotocol – [http]/1.1," June 1999. [RFC] 2616, http://www.ietf.org/rfc/rfc2616.txt.

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

[5] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, pp. 333–340, June 1975.

[6] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 3rd ed., 2007.

[7] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *CCS*, pp. 262–271, 2003.

[8] R. Smith, C. Estan, and S. Jha, "Xfa: Faster signature matching with extended automata," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 187–201, May 2008.

[9] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 339–350, 2006.

[10] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *ACM/IEEE ANCS*, pp. 145–154, 2007.

[11] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pp. 93–102, 2006.

[12] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *ACM CoNEXT*, pp. 1:1–1:12, December 2007.

[13] Y. Afek, A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "Mca2: Multi-core architecture for mitigating complexity attacks," in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, (New York, NY, USA), pp. 235–246, ACM, 2012.

[14] R. Smith, C. Estan, and S. Jha, "Backtracking algorithmic complexity attacks against a nids," in *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pp. 89–98, Dec 2006.

[15] A. Bremler-Barr and Y. Koral, "Accelerating multipattern matching on compressed http traffic," *IEEE/ACM Trans. Netw.*, vol. 20, no. 3, pp. 970–983, 2012.

[16] P. Rauschert, Y. Klimets, J. Velten, and A. Kummert, "Very fast gzip compression by means of content addressable memories," in *TENCON 2004. 2004 IEEE Region 10 Conference*, vol. D, pp. 391–394 Vol. 4, Nov 2004.

[17] J. Franklin, "Hardware accelerated compression," May 23 2006. US Patent 7,051,126.

[18] K. Ma and W. Chen, "Method and apparatus for efficient hardware based deflate," Dec. 11 2007. US Patent 7,307,552.

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

[20] T. Kida, M. Takeda, A. Shinohara, and S. Arikawa, "Shift-and approach to pattern matching in lzw compressed text," in *10th Annual Symposium on Combinatorial Pattern Matching (CPM 99)*, 1999.

[21] G. Navarro and M. Raffinot, "A general practical approach to pattern matching over ziv-lempel compressed text," in *10th Annual Symposium on Combinatorial Pattern Matching (CPM 99)*, 1999.

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

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

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

[25] A. Bremler-Barr, Y. Koral, and V. Zigdon, "Shift-based pattern matching for compressed web traffic.," in *HPSR*, pp. 222–229, IEEE, 2011.

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

[27] Y. Afek, A. Bremler-Barr, and Y. Koral, "Space efficient deep packet inspection of compressed web traffic," *Comput. Commun.*, vol. 35, pp. 810–819, Apr. 2012.

[28] M. Berger and B. Mortensen, "Fast pattern matching in compressed data packages," in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pp. 1591–1595, Dec 2010.

[29] Y. Sun and M. S. Kim, "Dfa-based regular expression matching on compressed traffic," in *Communications (ICC), 2011 IEEE International Conference on*, pp. 1–5, June 2011.

[30] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, pp. 419–422, June 1968.

[31] "PCRE - perl compatible regular expressions." http://www.pcre.org.

[32] M. Becchi, "Regular expression processor." http://regex.wustl.edu.

[33] "Alexa: The web information company," Dec 2011. http://www.alexa.com/topsites.

APPENDIX

Proof: We will prove Claim VI-A by induction, showing that it holds for each NFA construction step. Recall that each such step takes an operator and one or two operands and constructs an NFA part.

Induction base: The first NFA construction step may take as operand a single character. This expression by definition contains no complex states. The first operator may be either a Kleen closure or concatenation with an empty expression. The former case results in marking all states as complex, thus the above claim still holds. The latter case constructs a simple state. Still, since this state represents a string of length 1, the above claim also holds. An alternation requires two expressions thus it cannot be used at the first construction step. Therefore the induction base holds.

Induction step: Assume that the above claim holds for k construction steps. We prove for the $k+1$ construction step. Assume on the contrary that there is a state s, marked as simple, which after construction step $k+1$ has two input strings of different length that upon scan from s_0 terminate at s. Let s be part of expression S. We now inspect each operator. For a Kleen closure over expression S all its states would have been marked as complex. Since s is simple, this is a contradiction. For a concatenation of expression S to R, assuming there are at least two input strings with different length that lead to s, then r, the last state of R must have also at least two such input strings. In this case r should have been marked as complex prior step $k+1$. Thus, tall states of S including s should have been marked complex at step $k+1$, which again is a contradiction. The only option left is the alternation operator. Note that this operator does not change marks for states inside the operand expressions, rather it creates two additional states prior and after the expressions. Thus our assumption could take place only if s is a newly created state at step $k+1$. The new state before the expression accepts only an empty string, which is not the case for s. Therefore, s must have been the new state after the expressions. Since s is simple, both expressions' last state must be simple. The only case where two input strings with different lengths could have reach s from s_0 is if both expressions represent a string with different length. But in this case s would have been marked as complex, which is again a contradiction. Thus the claim holds for $k+1$ and the proof of the induction is complete. ■