Accelerating Regular Expression Matching over Compressed HTTP Traffic

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science (M.Sc.) Research Track in Computer Science

Submitted by Omer Kochba

Under the supervision of Prof. Anat Bremler-Barr & Dr. Yaron Koral

March 2016
Abstract

This work 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. 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 matching, taking advantage of information gathered when the traffic was initially compressed. HTTP compression is typically performed through the GZIP protocol, which uses back-references to repeated strings. 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 implementations and gains performance boosts 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.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>DFA Example</td>
<td>11</td>
</tr>
<tr>
<td>4.2</td>
<td>DFA Example</td>
<td>11</td>
</tr>
<tr>
<td>4.3</td>
<td>NFA Example</td>
<td>14</td>
</tr>
<tr>
<td>4.4</td>
<td>Basic automaton operators</td>
<td>17</td>
</tr>
<tr>
<td>4.5</td>
<td>DFA Example for positive/negative transitions</td>
<td>18</td>
</tr>
<tr>
<td>4.6</td>
<td>Data structures used for positive/negative transition marking</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Illustration of the Hybrid ARCH-DFA System architecture</td>
<td>24</td>
</tr>
<tr>
<td>6.1</td>
<td><em>Input-Depth</em> Cumulative Distribution</td>
<td>28</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Simulation of ARCH over Active States NFA ..................... 14
6.1 Rule-sets characteristics ........................................... 25
6.2 ARCH Performance .................................................. 26
List of Algorithms

1 Modified subset construction algorithm ........................................ 19
2 Marking positive/negative transitions ........................................ 20
3 ARCH-DFA Input-Depth Maintenance ......................................... 21
Chapter 1

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 inspects the packets’ payload in addition to their header. Recently, especially due to the proliferation of mobile devices with limited bandwidth, DPI components have had to deal also with compressed traffic. This adds an additional performance penalty of data decompression prior to the inspection. Finding an efficient solution is crucial as nowadays the majority of Web-sites use compression. HTTP compression is typically done with the 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. However, contemporary DPI engines are required to support also regular expressions.

This work 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 GZIP), which were in a sense “already scanned”. Extra care is taken to detect and handle delicate cases where the regular expression matches consecutive repeated and non-repeated strings (e.g., a pattern prefix followed

\[1\text{A recent report shows that 64.4\% of the Internet Web sites use HTTP compression. When looking at the top 1000 most popular sites, over 92\% of them use HTTP compression.}\]
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 Chapter 2). In case of regular expressions, every state may represent a wide set of strings with various lengths, as 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’ scans lead to the same state. To overcome this problem, we provide an algorithm that evaluates a new parameter called \textit{Input-Depth}, which relates to the length of the input’s shortest suffix that leads to the current state from the automaton’s root.

We provide an algorithmic framework called \textit{Acceleration of Regular expression matching over Compressed HTTP (ARCH)}, which uses the \textit{Input-Depth} parameter. We derive two system designs from this framework with respect to the two distinct DPI approaches, namely: \textbf{two phase inspection} — string based pre-filtering accompanied by an NFA scan for regular expression matching, and a \textbf{single pass inspection} — DFA-based regular expression matching. Our experiments show that the first 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 IPS \cite{2}. 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 to 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 gains a performance boost of 3.3 times.

This work makes the following contributions:

1. A study of the challenges of regular expression matching over compressed traffic.
2. The first algorithmic framework that accelerates regular expression matching over compressed traffic.
3. A generic algorithm to accelerate regular expression matching over NFA based automata.

4. A generic algorithm, with two variants, to accelerate regular expression matching over DFA based automata.

5. A system setup which allows handling large, complex rule-sets which cause state-space explosion.

6. A live setup which experiments both the correctness as well as the performance benefits of using the different algorithms.

7. A discussion of additional possible applications for this algorithm.
Chapter 2

Background

2.1 Compressed HTTP

Compressed HTTP (namely, content coding for HTTP) is a standard method of HTTP 1.1 [3]. The standard describes the following content codings: GZIP, DEFLATE and COMPRESS. Practically, only the former two schemes are used. Since GZIP is a variant of DEFLATE, this work 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 replacing repeated strings within the last 32KB of uncompressed data 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 data size 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 encodes the literals and pointers using Huffman coding. Specifically, the algorithm in this work is based on the characteristics of the LZ77 compression.
2.2 Deep Packet Inspection Algorithms

DPI involves processing of the packet payload to identify occurrences of a set of pre-defined patterns. These patterns are expressed as either string or regular expressions. 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 the input size.

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

DFAs scan the input in linear time, but use 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 our design for large pattern sets is inspired by Hybrid-FA [12].

NFAs, on the other hand, use linear space but have worst-case quadratic time complexity [9], which may be exploited by an adversary for denial-of-service attack (DoS) on the DPI element itself [13, 14]. In NFAs, 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 state $s$ with label $b$). A common implementation maintains a set of active states $S$; for each input byte $b$ this set is recomputed by traversing all $b$-labeled edges from every state $s \in S$.

In practice, security tools with a large rule-set such as Snort [2], use a pre-filter based solutions. The pre-filter performs string matching on extracted strings from the regular expressions. When all strings of a specific rule match, a regular expression scan is
invoked using an NFA. The pre-filter can be accelerated over compressed traffic based on prior solutions for string matching (as described in the sequel), while the NFA part can be accelerated using our ARCH-NFA solution.

2.3 String Matching Over Compressed Traffic

LZ77 is an adaptive compression as each symbol is determined based on its preceding 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 of the decompressed data. The idea behind the acceleration algorithm is to use the information gathered by the decompression phase to skip scanning significant parts of the data. An LZ77 pointer represents a repeated string that was already scanned. Therefore, it is possible to skip scanning most of the it without missing any pattern. Still, tracking previous matches does not suffice due to cases 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 occurs at the repeated string ‘bcdnb’, there are matches occurring on both its left and right borders.

This problem is tackled by the DFA-based ACCH algorithm [15], which is limited to string matching. 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 contain 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. Its values are determined by DFA-Depth\(s\) — the length of the shortest path from root to state \(s\). There are three possible status values:

- MATCH: a pattern was matched (the match ends in the scanned byte).
• UNCHECK: $\text{DFA-Depth}(s)$ is below a threshold parameter $\text{CDepth}$ (in practice, $\text{CDepth}$ is set to 2).

• CHECK: otherwise.

ACCH uses the $\text{Status-Vector}$ in the following manner:

Left Border Scan — Upon processing a repeated string, ACCH scans $j$ bytes until reaching a state $s$ where $j \geq \text{DFA-Depth}(s)$. 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 corresponding $\text{Status-Vector}$ values of the referred string, starting at its $(j + 1)^{\text{th}}$ byte. If some byte at position $m$ has a MATCH value, ACCH checks the length of that previously matched string. If it is $m$ or more then the previously matched string was not repeated entirely. Otherwise, ACCH reports a matched pattern as in the referred string and marks that byte by a MATCH status.

Right Border Detection — ACCH determines the position to start inspection again such that no pattern (whose prefix is part of the repeated string suffix) is missed. This is done by estimating the $\text{DFA-Depth}$ 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 corresponding referred string. Note that since ACCH skips only after it ensures that there is no pattern whose prefix started prior to the repeated string, the value of the estimated $\text{DFA-Depth}$ could be only equal or greater than its actual value (namely, setting a byte as CHECK while it should be UNCHECK). Such a mistake leads to minor reduction in the amount of skipped bytes but never to a miss-detection.
Chapter 3

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 Chapter 2, HTTP compression uses LZ77 compression. There are various works 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 proposals 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] deal with pattern matching over LZ77. However, their proposed algorithm matches only a single pattern and requires two passes over the compressed text (file), which does not comply with the ‘on-the-fly’ processing requirement of network applications.

The ACCH algorithm [15], discussed in details in Chapter 2, was the first 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 [3] 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 to regular expression matching, thus SPC cannot be used in our case. The SOP [27] algorithm minimizes the memory footprint required by the ACCH data structures. It may be combined with our algorithm, with few adaptations, to save space. Finally, Berger and Mortensen [28] provide a hardware scheme for the decompression phase, but do not deal with the scanning of the compressed data itself.

All the above mentioned works are limited to string matching rather than to devices based on regular-expression–matching. Sun et al. [29] suggested a method to perform 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 traverses to deeper DFA states. Furthermore, an attacker may easily craft an input that causes the DFA traversal into areas that the algorithm of Sun et al. fails to support. Robustness against such attacks is crucial as IDSs are a preferred target for denial-of-service attacks.
Chapter 4

The ARCH Framework

This chapter 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 Chapter 2), which works only for string matching. 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 happens as a result of an alternation between different-length expressions or a Kleene closure (namely, ‘*’ or ‘+’). An example of a state with an ambiguous depth in the presence of an alternation operator is depicted in Fig. 4.1, which represents an automaton accepting the pattern ‘(apple|pear)s’. The DFA-Depth($s_5$) value is 4 since it is the shortest path length from $s_0$ to $s_5$. 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 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. 4.2. The DFA is for ‘ab+c+’ and an input of ‘bbbbbcxa{8,7}’.
Figure 4.1: DFA for ‘(apple|pear)s’. Failure transitions are marked with red dashed arrows. Failure transitions to states with DFA-Depth 0 and 1 are omitted for clarity.

Figure 4.2: DFA for ‘ab+c+’. The right table indicates the input data and its DFA-Depth as derived from the automaton while scanning the input.

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 the 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 $X$ with the automaton. Let $Input-Depth(X, s)$ be the length of the shortest suffix of $X$ 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,
the ARCH framework uses the ACCH algorithm to calculate possible scan skipping by replacing the DFA-Depth parameter with the 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.

4.1 Input-Depth Calculation for NFA-Based Implementations

Recall that one possibility to provide regular expression matching is to use NFAs, which are usually compact but slow data structures. 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 [4.2]). Moreover, NFAs are currently used by the popular Snort IPS, and therefore, accelerating their operations has a merit on its own. A commonly used NFA implementation which we refer to is Active-States NFA. This implementation maintains a set of active states. During each step, if there is a valid transition from an active state $s$ to one or more states denoted by the set $S'$, then this state is replaced in the active-states set by the set $S'$. In this case we refer to $s$ as the predecessor of each state in $S'$. If there are no such transitions, the state simply becomes inactive, i.e. removed from the active-states set.

ARCH maintains an Input-Depth value for each element in the active-states set. Namely, given an input prefix $Y$, an active state $s$, an input byte $b$, and a subsequent active state set $S'$ such that there is a transition between $s$ and each state $s'$ in $S'$ with a label $b$, the value of $\text{Input-Depth}(Yb,s')$ is set to $\text{Input-Depth}(Y,s)+1$. The only exception is at
the start state of the NFA (which is always active) and for which \( \text{Input-Depth}(Y,s_0) \) is always zero. Incrementing the \( \text{Input-Depth} \) by 1 is based on the following claim:

Claim 1. The \( \text{Input-Depth} \) of an active state can only grow by 1 compared to its predecessor.

Proof. Given an NFA, its active state set \( S \), and an input byte \( T_{i+1} \). For each state \( s' \), that is a new state added to the active-states set, there exists at least one labeled transition from some state \( s \) which is already in the active states set, denoted by \( T_{i+1} \). Assume on the contrary that \( \text{Input-Depth}(T_{i+1},s') > \text{Input-Depth}(T_i,s) + 1 \). Thus inspecting a suffix of only \( T_{i+1} \) from the active state \( s \) would end up in a set of states \( S'' \) that does not include \( s' \), in contrary to the assumption that \( T_{i+1} \) leads to \( s' \). Therefore this is a contradiction, hence \( \text{Input-Depth} \) grows at most by 1 compared to its predecessor after each step. \( \square \)

We define \( A(Y) \) as the active-states set after scanning the input \( Y \). We define the \( \text{Global-Input-Depth}(Y) \) as the longest relevant suffix of the input. We determine \( \text{Global-Input-Depth}(Y) \) by choosing the maximal \( \text{Input-Depth}(Y,s) \) where \( s \) belongs to \( A(Y) \). When ARCH needs to determine the left or right border of a pointer it uses the \( \text{Global-Input-Depth}(Y) \). This is based on the following claim:

Claim 2. The \( \text{Global-Input-Depth}(Y) \) is bound by the maximal \( \text{Input-Depth} \) of the active states set.

Proof. Assume on the contrary that the \( \text{Global-Input-Depth}(Y) \) denoted by \( x \) is larger than the maximal \( \text{Input-Depth} \) of \( A(Y) \) such that \( x > \max(\text{Input-Depth}(Y,s) \mid s \in A(Y)) \). Thus, for the current input, there is a suffix of length \( x \) that resulted in at least one state \( s' \) for which \( \text{Input-Depth}(Y,s') = x \). Then, \( s' \) does not belong to \( A(Y) \) because otherwise \( x = \max(\text{Input-Depth}(Y,s) \mid s \in A(Y)) \), contrary to the assumption that \( x > \max(\text{Input-Depth}(Y,s) \mid s \in A(Y)) \). Therefore this implies that the construction algorithm is incorrect, which contradicts the basic Thompson algorithm for pattern matching \([30]\). Therefore the global NFA \( \text{Input-Depth} \) is bound by the maximal \( \text{Input-Depth} \) of the active states set. \( \square \)
Since the \textit{Input-Depth} of each active state can only increase by at most 1 (claim 1), the \textit{Global-Input-Depth}(Y) of the NFA increases by at most 1 after inspection of a single byte.

Fig. 4.3 depicts an NFA for the patterns \textquote{zabcfg} and \textquote{z[^\n]*fg}. The execution of ARCH for input \textquote{zf\nzabcfe} and the NFA are illustrated in Table 4.1.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Input Character} & \textbf{Active States} & \textbf{Input-Depth Set} & \textbf{Max Input-Depth} \\
\hline
z & \(s_0, s_1, s_7\) & 0, 1, 1 & 1 \\
\textbackslash n & \(s_0\) & 0 & 0 \\
\textbackslash n & \(s_0, s_1, s_7\) & 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 \\
\hline
\end{tabular}
\end{table}

\section{4.2 Input-Depth Estimation for DFA-Based Implementations}

The task of \textit{Input-Depth} calculation is more challenging when a DFA is used for regular expression matching. This is because, unlike NFA, a DFA transition may result either in increasing the \textit{Input-Depth} (by one) or decreasing the \textit{Input-Depth} (by any value).

In this section, we provide an \textit{upper bound} on the \textit{Input-Depth} using two methods: by \textit{simple and complex states} and by \textit{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.

### 4.2.1 Estimation based on Simple and Complex States

As noted above, Input-Depth cannot be always derived from DFA-Depth. Still, there are cases where 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, in the DFA of Fig. 4.2 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 $s$ have the same length. All other states are complex.

Given the above definition, the following claim holds for complex states:

**Claim 3.** If a state $s$ is complex then there is at least one path, whose scan from $s_0$ terminates at $s$, where Input-Depth can be inflated (i.e. there are bytes or strings that can be added to the input text such that the Input-Depth will grow while the DFA-Depth remains the same).

**Proof.** Given a DFA and the current state $s$, which is complex. Assume on the contrary that a scan from $s_0$ which terminates at $s$ cannot be inflated. Then there exists no transition along all possible paths from $s_0$ to $s$ that may inflate the Input-Depth, i.e. increase the Input-Depth without increasing the DFA-Depth. However, by the definition of a complex state, $s$ is only complex if it can be reached from some other complex state $t$ which was marked complex because of a backward transition to the current path that may inflate the Input-Depth. Therefore, this is a contradiction and therefore there exists at least one path whose scan from $s_0$ terminates at $s$, where Input-Depth can be inflated. 

\[\square\]
Input-Depth Estimation: According to claim 3 above, one cannot estimate Input-Depth of a complex state by the automaton properties solely, as the upper bound would always be infinity. To provide a better upper bound we estimate Input-Depth of complex states online per each scan step. Since per input character the Input-Depth can grow by at most 1 (as shown in claim 4 below), we estimate Input-Depth for complex states by incrementing it by 1. Therefore, 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. We note that by using this method Input-Depth may be over-estimated for complex state. However, this is corrected whenever the automaton falls back to a simple state where the the Input-Depth is derived exactly.

Claim 4. Inspection of a single byte increases Input-Depth by at most 1.

Proof. Let the last two transitions of inspecting \( T_{1...i+1} \) lead to states \( s \) and \( s' \) respectively. Assume on the contrary that \( \text{Input-Depth}(Y, s') > \text{Input-Depth}(s) + 1 \). Thus inspecting the input character of only \( T_{i+1} \) from state \( s \) would end up in a state \( s'' \neq s' \). Thus for input byte \( T_{i+1} \) from state \( s \) there are two transitions to two different states \( s' \) and \( s'' \), which is not possible in DFA. Therefore this is a contradiction, hence Input-Depth grows at most by 1 after each step. \( \square \)

Detection of Simple/Complex states: 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, \( \epsilon \)-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 NFA construction, and then we transfer these marks to the final DFA. The NFA construction is defined over the three basic regular expression operators as in Fig. 4.4. The operands are regular expressions \( R \) and \( S \). All states are marked as simple by default and stay that way unless otherwise specified. Complex states are marked according to each regular expression operator as follows:

- **Kleene Closure** (Fig. 4.4(a)): Mark all states as complex.
4.2.2 Estimation based on Positive and Negative Transitions

In the previous section we estimated an upper bound for Input-Depth using simple and complex states. Although the method is intuitive and efficient, it may overestimate Input-Depth in some cases, thus causing redundant scans. In this section we provide a method to estimate a more accurate upper bound on the value of Input-Depth. It is based on transitions rather than on states.

The positive and negative transition technique 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. 4.5 shows that the transition from $s_x$ to $s_y$ should increase the Input-Depth, the transition between $s_z$ to $s_y$ should leave the Input-Depth unchanged, and the
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 $c$-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 $c$-label($s$) $\in L(s)$ such that $c$-label($s$).length $\leq l(s).length$ for any $l(s) \in L(s)$. The algorithm is depicted in Alg. 1. We refer to the second stage of the Thompson construction (as described at Section 4.2.1), which is the subset-construction that constructs a DFA from an NFA. The $c$-label($s$) parameter is determined by integrating a line into the subset-construction algorithm (as in Alg. 1 Line 8). The algorithm uses the following basic NFA operations:

- $\epsilon$-closure($T$) — The set of NFA states reachable from each NFA state $s$ in set $T$ on $\epsilon$-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 $t$ (Line 7). When a
transition from the DFA state $t$ results in a DFA state $u$, which was never visited
before, the algorithm creates a $c$-label($u$) by concatenating $c$-label($t$) and the transition
label $a$ (Line 8). Note that there may be more transitions that lead to DFA state $u$
along the construction. Still, since BFS is used for the NFA traversal, and the $c$-label($u$)
was generated upon the first transition that reaches $u$, $c$-label($u$) must be no longer than
any other label in $L(u)$, as required.

**Algorithm 1:** Modified subset construction algorithm. The symbol $\oplus$ means concatenation.

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 $c$-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 $c$-label($t$) $\oplus$ $a$ that leads from the root to
$u$. Therefore, a string such as $c$-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. This procedure is possible due to the following claim:

**Claim 5.** For every $c$-label($t$) there exists a non-empty set of states $R$ in the Anchored-NFA where $R=\epsilon$-closure(move($c$-label($t$)).

**Proof.** Given a DFA, and its set of states $S$. For every state $s$ in $S$ there is a $c$-label value denoted by $c$-label($s$). Assume on the contrary that there exists a state in the DFA $t$, its $c$-label value is $c$-label($t$), and that the set of states $R=\epsilon$-closure(move($c$-label($t$)) is empty. By definition, $c$-label is set during the subset-construction algorithm by traversing the NFA in breadth-first search (BFS). If the state $t$ was marked with $c$-label($t$), then there exists some state $u$ in the NFA which corresponds to it. Since state $u$ was reached by an input of the series of bytes that is equal to $c$-label($t$) then when simulating the same byte sequence on the corresponding Anchored-NFA, $R$ cannot be empty because the Anchored-NFA is equivalent to the original NFA for patterns that exist in the beginning of the input (as in this case). Therefore this is a contradiction, and for every $c$-label($t$) there exists a non-empty set of states $R$ in the Anchored-NFA where $R=\epsilon$-closure(move($c$-label($t$)).

\begin{verbatim}
foreach state t in Dstates do
    R=Anchored-NFA(move(c-label(t)));
    foreach input symbol a do
        u=Dtran[t,a];
        if $\epsilon$-closure(move($R$, $a$))\neq \emptyset then
            Dtran[t,a].positive=true;
        else
            Dtran[t,a].positive=false;
        end
    end
end
\end{verbatim}

**Algorithm 2:** Marking positive/negative transitions

For example, consider the patterns ‘ab+c+d’ and ‘bc+e’. The resulting DFA from our modified subset construction is depicted in Fig. 4.6(a) and its full-matrix representation along with its calculated $c$-label($s$) values is depicted at Fig. 4.6(c). After running Alg. 2 using the DFA and the Anchored-NFA (depicted in Fig. 4.6(b)) all negative,
positive (underline) and loop (circled) transitions are marked. All negative transitions are marked with dashed red arrows at Fig. 4.6(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
2
3
4
5
6
7
8
9
10
11

Algorithm 3: ARCH-DFA Input-Depth Maintenance

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

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.6.png}
\caption{Data structures used for the positive/negative transition marking for pattern set: \{‘ab+c+d’, ‘bc+e’\}.}
\end{figure}
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.
Chapter 5

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 descendant 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 ‘AdminServlet’, Input-Depth cannot decrease since any character can be part of the pattern and may be followed by either ‘userid’ or ‘adminurl’. Thus, ARCH cannot skip scanning characters in such cases as it never finds the left border of a pointer (see Chapter 4). 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 Chapter 6) 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 Chapter 6. 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, however, future pattern sets or implementations might be different.

ARCH is inspired by the design of Hybrid-FA by Becchi et al. [12]. ARCH system breaks the regular expressions into a prefix set of all expressions, which are represented by a single prefix ARCH-DFA (which is analogous to the head-DFA in [12]), and a set of tail ARCH-DFAs that represent the entire regular expressions set. The prefix ARCH-DFA is always active, while only a small part of the tail ARCH-DFAs may be active. This way the resulting hybrid automaton has a small memory footprint as compared to the corresponding DFA, with the penalty of traversing several automata 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 benefits as depicted in Fig. 5.1.
Chapter 6

Experimental Results

We evaluate the performance benefits of ARCH on rule-sets from the Snort IPS. The Snort24, Snort31, and Snort34 rule-sets were taken from [32]. The Snort135 rule-set was extracted from Snort’s "web-client.rules" (February 2014). These rules relate to Web traffic. Table 6.1 summarizes the basic characteristics of the rule-sets used in the experiments.

6.1 Data Set

The data set contains 2301 compressed HTML pages, downloaded from 500 popular sites taken from the Alexa Web-site [33]. The data size is 358MB in uncompressed form and 61.2MB in compressed form. Since the algorithm is based on repeated byte sequences, it may only perform as well as the compression ratio of the input data and is therefore mainly aimed at highly-compressible inputs such as text.

Table 6.1: Rule-sets characteristics

<table>
<thead>
<tr>
<th>Rule Set</th>
<th>Snort24</th>
<th>Snort31</th>
<th>Snort34</th>
<th>Snort135</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Rules</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>% Rules with Kleene Closure</td>
<td>37.5</td>
<td>41.9</td>
<td>38.2</td>
<td>89.6</td>
</tr>
<tr>
<td>% Rules with Char Ranges $\geq$ 50</td>
<td>50</td>
<td>48.4</td>
<td>41.1</td>
<td>51.9</td>
</tr>
</tbody>
</table>
6.2 ARCH performance

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 skipped using ARCH out of the total size of the input data. We define $R_t$ as the saved scan time ratio — ARCH’s running time compared to the baseline algorithm’s running time.

ARCH-NFA was implemented using active states NFA as its baseline algorithm, as described in Section 4.1. Table 6.2 shows statistics regarding its performance. The average skip rate $R_s$ is 77.99%, which results in a significant performance improvement $R_t$ of 77.21%. Compared to the overall performance, the overhead of ARCH-NFA is less than 1%. The overall processing time of ARCH-NFA is relatively long compared to the DFA based implementations (40 times longer) and is therefore less preferred. On the other hand, the space requirements of ARCH-NFA are 18 times lower than those of ARCH-DFA. The Snort135 rule-set was not tested with ARCH-NFA due to the high memory requirements of representing many rules in a single automaton. This work does not aim to solve the low performance of NFA based algorithms, instead it aims to show that the ARCH algorithm fits and may be used by both approaches.

ARCH-DFA was implemented using A-DFA [10] as its baseline compression algorithm. The implementation performs Input-Depth estimation based on both of the methods described in Section 4.2. The results shown below are based on the simple and complex states method which is described in Section 4.2.1. This estimation has a lower overhead while the benefit gained by the more precise estimation of the positive/negative

<table>
<thead>
<tr>
<th>Rule Set</th>
<th>Snort24</th>
<th>Snort31</th>
<th>Snort34</th>
<th>Snort135</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA Characters Skipped</td>
<td>79.44%</td>
<td>75.86%</td>
<td>78.66%</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>79.29%</td>
<td>74.56%</td>
<td>77.78%</td>
<td>-</td>
</tr>
<tr>
<td>Time Saved</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFA Setup</td>
<td>Single</td>
<td>Single</td>
<td>Single</td>
<td>Hybrid</td>
</tr>
<tr>
<td></td>
<td>79.04%</td>
<td>75.64%</td>
<td>78.38%</td>
<td>78.46%</td>
</tr>
<tr>
<td></td>
<td>70.97%</td>
<td>67.46%</td>
<td>69.14%</td>
<td>70.08%</td>
</tr>
<tr>
<td>Characters Skipped</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time Saved</td>
<td>168603</td>
<td>1333</td>
<td>8000</td>
<td>77801</td>
</tr>
<tr>
<td>Number of Matches</td>
<td>38.46</td>
<td>45.45</td>
<td>40</td>
<td>-</td>
</tr>
<tr>
<td>Run-time NFA/DFA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
transitions method is not significant on our data sets. However, in order to prove the theoretical advantage of the positive/negative transitions method for certain edge cases, we created a synthetic data set which includes these edge cases. The result shows an improvement of approximately 18% in the average Input-Depth which allows the ARCH algorithm to skip more bytes and reduce the scan time.

We implemented two system setups 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. Table 6.2 shows statistics regarding the performance of ARCH-DFA. ARCH-DFA achieved an average skip rate $R_s$ of 77.69% and a performance boost $R_t$ of 69.19%. In this case the overhead is more notable and is almost 11%. As discussed in Chapter 5, this is mainly 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 Chapter 5. 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 $R_t$ is 69.41%. The overhead in this case is also 11%. We note that since the performance improvement of the algorithm depends on the properties of both the input string and the regular expressions in the pattern set, the method may offer a lower performance gain when detecting patterns that contain a short string prefix followed by Kleen Closure.
Figure 6.1: Input-Depth Cumulative Distribution
Chapter 7

Additional Applications

Beside regular expression matching over compressed traffic, *Input-Depth* is important for many additional applications. In this chapter we describe three such applications:

1. Extraction of a matched pattern
2. Patterns across packet fragmentation
3. Efficient Pre-Filtering

7.1 Extraction of a matched pattern

A surprising fact is that there is no straightforward method to extract the string that relates to a matched pattern after detecting a match in the automaton (by reaching a matching state) without having to rescan 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.
7.2 Patterns across packet fragmentation

Studies have shown that Intrusion Prevention Systems may be evaded by using packet fragmentation. Such attacks are implemented by splitting the malicious pattern across several packets. A naive approach to handle such threats is to completely re-assemble the packets into one stream. This method may require high memory usage and severely impact performance in case of long matches across several packets. Therefore, an important 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.

7.3 Efficient Pre-Filtering

Another important application is efficient pre-filtering for intrusion prevention systems. Systems such as Snort perform pre-filtering on a simpler automaton in order to identify the existence of a prefix of a pattern in the current packet. If the pre-filtering identifies such a match, the entire packet is re-scanned against the relevant regular expressions. Instead, an efficient pre-filtering method may use Input-Depth in order to scan only the relevant bytes in the second phase instead of having to re-scan the entire packet. This has an even larger benefit in the case where the match spans across several packets.
ARCH started as an effort to adapt prior art of string matching over compressed traffic to the regular expression matching domain. Along this process we uncovered a 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 either moderate simple pattern sets and large complex ones. Three architectures are proposed: ARCH-NFA, ARCH-DFA and the Hybrid-ARCH-DFA.

This work not only provides the theoretical background behind the different methods, it also provides a setup that tests the performance benefits of the provided algorithms on real life Web traffic.

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 many other applications for which it is important.
Bibliography


[27] Yehuda Afek, Anat Bremler-Barr, and Yaron Koral. Space efficient deep packet
2012. ISSN 0140-3664. doi: 10.1016/j.comcom.2012.01.017. URL http://dx.doi.
org/10.1016/j.comcom.2012.01.017.

[28] M.S. Berger and B.B. Mortensen. Fast pattern matching in compressed data pack-
2010. doi: 10.1109/GLOCOMW.2010.5700208.

[29] Yan Sun and Min Sik Kim. Dfa-based regular expression matching on compressed
traffic. In *Communications (ICC), 2011 IEEE International Conference on*, pages

363387. URL http://doi.acm.org/10.1145/363347.363387.


topsites.

[34] T. Ptacek and T. Newsham. Insertion, evasion and denial of service: Eluding
התקציר

dחוסה בעזרת ביטויים רגולריים. HTTP

עבודה זו עוסקת ב прожון של ביטויים רגולריים. התחלתי, את התעבורה הדחוסה ברשות, במציאת מבית ביטויים רגולריים, נuumמאמים לגלות עדות מתמדת על מתןلزمאותữaוד

הتعبורה. שיתוף הפעולה של ביטויים רגולריים או שיתוף הפעולה של תבניות יocalyיד את מגדלת הזמן בשירותים. בפיזור, נמשכת תכנית באמצעות ביטויים רוגieriים גלקלי

היכולה לצבוק מספר זאוז של תבניות בזרועה防晒ה את התחלקה

הخوف של ביטויים מוסון זה. התשתית המנצלה מיועד לשאר תכנית בזואります, HTTP

דחוס לבר בyardת אלגוריתמים. זאש משותף בצבועה. התעבורה

אוחרו על מתן לייז מחרוזות בזאחר על עבום בימי עד ברז מתאיצן קוס .יור

של חומרים. האלגוריתמים שלול מבוסע על שושב המסר הפיקסליים של חומרי ( kodimks)

רלוונטיים, עבר cls, ונשא ע Yöן לשון חקミニי של הבדין. נlararası נושא בזואיס

הצבועה-אוחרו, רק חומרי אלו אוכזבי לילה חракти חסובן כל חומר על חקמין

נרציבים ממומדות על שושב מספים של חומרי פורטפוליו. על ידי כט נ iota לחשוף ומיה

רב במקהל הסיסק אומן אלגוריתמים אוצי שאחר תורא על תשתית הפרות.

בנוסף, אנחוג מורארו בזוארשחביקים עם התשתית של הדום עבודה ומעמידים סוזים של

הטсложн סירק אוש מובסס על אוטומטיים מוסון לא DFA וא NFA. בזאי המכרזים

הטประสית של הדום מוספקת שיפור ביצועים של מעלה מ70%. בזוארש, חינית חוושמך

בצורה להובטח-aheadת פוליפים ציוד בזאמה ומסניפה כים חלומית מחרוזות הרלוונטיים לאחר

יחוי, עוד.
המרכז הבינתחומי הרצlia
בין"ס אפי ארי למדעי המחשב

האצתחיפוש מחזורות בוצעות בוירוס ביטויים

רגולרים מעל גבי תשבורות

розו

scssx עבוזת גמר המmisión של חלב מתודירישות לבראשית תואר

מוסכמת במשלול מחקרי במדעי המחשב

על ידי עומר קובל

העבורה בקשתה בちょותם פרופ' ענת ברמל-בר ור"ר ירון קורל

2016