Chapter 10
Fast Packet Pattern-Matching Algorithms

Fang Yu, Yanlei Diao, Randy H. Katz, and T.V. Lakshman

Abstract  Packet content scanning at high speed has become extremely important due to its applications in network security, network monitoring, HTTP load balancing, etc. In content scanning, the packet payload is compared to a set of patterns specified as regular expressions. In this chapter, we first describe the typical patterns used in packet-scanning applications and show that for some of these patterns the memory requirements can be prohibitively high when traditional matching methods are used. We then review techniques for efficient regular expression matching and explore regular expression rewrite techniques that can significantly reduce memory usage. Based on new rewrite insights, we propose guidelines for pattern writers to make matching fast and practical. Furthermore, we discuss deterministic finite automaton (DFA) link compression techniques and review algorithms and data structures that are specifically designed for matching regular expressions in networking applications.

10.1 Motivation

Packet content scanning (also known as Layer-7 filtering or payload scanning) is crucial to network security and network monitoring applications. It is useful for detecting and filtering packets containing worms and other attack code, and also for...
newly emerging edge network services. Examples of the emerging edge network services include high-speed firewalls, which protect end hosts from security attacks; HTTP load balancing, which smartly redirects packets to different servers based on their HTTP requests and Extensible Markup Language (XML) processing, which facilitates the sharing of data across different systems.

In packet-scanning systems, the payload of packets in a traffic stream is matched against a given set of patterns to identify specific classes of applications, viruses, protocol definitions, and so on. When viruses and worms were simple, patterns could be expressed using simple strings format, e.g., “GetInfo\x0d” is the signature for a back door attack [3]. However, as viruses and worms became more complex, they rendered simple pattern-matching approaches inadequate for sophisticated payload scanning. For example, polymorphic worms make it impossible to enumerate all the possible signatures using explicit strings. Regular expressions appear to be a suitable choice for these patterns due to their rich expressive power. Consider a regular expression, “Entry/file/\[0-9\]\{\[1,\]//\.*\x0Aannotate\x0A”, for detecting a Concurrent Versions System (CVS) revision overflow attack [12]. This pattern first searches for a fixed string pattern “Entry/file/” followed by 71 or more digits or dots, then a fixed pattern “/” followed by some arbitrary characters (.*) , and finally the pattern “\x0Aannotate\x0A”. Obviously, it is very hard to enumerate this type of attack using fixed string patterns.

As a result, regular expressions are replacing explicit string patterns as the pattern-matching language of choice in packet-scanning applications. In the Linux Application Protocol Classifier (L7-filter) [12], all protocol identifiers are expressed as regular expressions. Similarly, the SNORT [3] intrusion detection system has evolved from no regular expressions in its rule set in April 2003 (Version 2.0) to 1131 out of 4867 rules using regular expressions as of February 2006 (Version 2.4). Another intrusion detection system, Bro [1], also uses regular expressions as its pattern language.

In this chapter, we present a number of regular expression pattern-matching schemes. We begin with a brief introduction to regular expressions with simple examples in Section 10.2. We briefly survey traditional regular expression matching methods in Section 10.3. We then analyze the special characteristics of typical regular expression patterns used in network scanning applications in Section 10.4. We show that some of the complex patterns lead to exponential memory usage or low matching speed when using the traditional methods. Based on this observation, we propose rewrite rules for two common types of complex regular expressions in Section 10.5. The rewrite rules can dramatically reduce the sizes of resulting deterministic finite automata (DFAs), making them small enough to fit in high-speed memory. In Section 10.6, we review DFA compression techniques that can further reduce memory consumption. Finally, in Section 10.7, we discuss some advanced DFA processing techniques developed specifically for high-speed router implementation.
10.2 Introduction to Regular Expressions

A regular expression describes a set of strings without explicitly enumerating them. Table 10.1 lists the regular expression operators used to describe patterns within packets. An anchor ("\[\text{^}\]) is used when a pattern must be matched at the beginning of the input. The ‘[’ operator denotes the OR relationship. The ‘.’ operator is a single-character wildcard; in other words, any character can be matched using ‘.’. The ‘?’ operator is a quantifier representing zero or one instance of the previously specified pattern, ‘+’ operator stands for at least one, whereas ‘*’ denotes zero or more. “{}” can be used to define more specific occurrence restrictions. For example, “{3, 5}” stands for repeating three to five times. We can also use “[[]” to define a class; characters inside the brackets form a class with OR relationships between them. When ‘\’ appears in “[[]”, it has the special meaning of exception. For example, “[\[]\text{n}]” denotes anything but the return key.

Let us now consider a more complex regular expression that is used for detecting Yahoo messenger traffic: “\(\text{^} (\text{ymsg}\text{y}pns|\text{yhoo})\text{.j\}ypns|\text{yhoo})\text{:q:q:q:q:q:q:q}\text{:\text{[lwt]}:ETX}\text{nxc0nxc80}\).” According to the analysis by Venkat [15], Yahoo messenger commands start with ymsg, ypns, or yhoo. Therefore, this pattern first identifies any of these three strings “\(\text{ymsg}\text{y}pns|\text{yhoo})\).” The next seven or fewer bytes contain command length and Version information that varies among packets. So this pattern ignores those by using “.:.:.:.:.:.:.”. Then it identifies a letter l, w, or t using [lwt]. l stands for “Yahoo service verify”, w denotes “encryption challenge command”, and t represents “login command”. This pattern ends with ASCII letters c0 and 80 in the hexadecimal form because \text{nxc0nxc80} is the standard argument separator in hexadecimal notation.

10.3 Traditional Regular Expression Matching Schemes

Regular expression matching is a well-studied subject in Computer Science. In this section, we briefly review the traditional approaches to the general regular expression matching problem.

Table 10.1 Features of regular expressions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Pattern to be matched at the start of the input</td>
<td>^AB means the input starts with AB. A pattern without “\text{^}”, e.g., AB, can be matched anywhere in the input</td>
</tr>
<tr>
<td></td>
<td>OR relationship</td>
<td>A\text{</td>
</tr>
<tr>
<td>.</td>
<td>A single-character wildcard</td>
<td>A. matches any two-character string starting with A</td>
</tr>
<tr>
<td>?</td>
<td>A quantifier denoting one or less</td>
<td>A? denotes A or an empty string</td>
</tr>
<tr>
<td>+</td>
<td>A quantifier denoting one or more</td>
<td>A+ denotes at least one letter A</td>
</tr>
<tr>
<td>*</td>
<td>A quantifier denoting zero or more</td>
<td>A* means an arbitrary number of As</td>
</tr>
<tr>
<td>{}</td>
<td>Repeat</td>
<td>A{100} denotes a sequence of 100 As</td>
</tr>
<tr>
<td>[ ]</td>
<td>A class of characters</td>
<td>[lwt] denotes a letter l, w, or t</td>
</tr>
<tr>
<td>[[]</td>
<td>Anything but</td>
<td>[\text{\text{n}}] denotes any character except \text{n}</td>
</tr>
</tbody>
</table>
Finite automata are a natural formalism for regular expression matching. There are two main categories: deterministic finite Automaton (DFA) and Nondeterministic Finite Automaton (NFA). This section provides a brief survey of existing methods using these two types of automata.

10.3.1 DFA

A DFA consists of a finite set of input symbols, denoted as $\Sigma$, a finite set of states, and a transition function $\delta$ [8]. In networking applications, $\Sigma$ contains the $2^8$ symbols from the extended ASCII code. Among the states, there is a single start state and a set of accepting states. The transition function $\delta$ takes a state and an input symbol as arguments and returns a state. A key feature of DFA is that at any time there is at most one active state in the DFA.

Figure 10.1 shows a simple DFA for regular expression $((A|B)C|(A|D)E)$, which matches string $AC$, $BC$, $AE$, or $DE$. If the given string is $BC$, it will first go to State 1 based on character $B$, then it will arrive at the final accept state (State 2) based on character $C$. Given another input starting with $A$, the DFA will first go to State 5. Depending on whether the next input is $B$ or $E$, it will transition to the corresponding accept state, that is, State 2 or State 4. If the next input is neither $B$ nor $E$, DFA will report the result of no-match for the given string.

10.3.2 NFA

An NFA is similar to a DFA except that the $\delta$ function maps from a state and a symbol to a set of new states. Therefore, multiple states can be active simultaneously in an NFA.

Figure 10.2 shows the NFA for the previous example $((A|B)C|(A|D)E)$. Unlike a DFA, for the NFA given an input starting with $A$, two states will be active at the same time (State 1 and State 3). State 1 means we have already seen the prefix

Fig. 10.1  A DFA example
pattern \((A|B)\), now waiting for the character \(C\). State 3 means we have seen \((A|D)\), now waiting for the character \(E\). Depending on the next input character, the NFA will go to State 2 if given \(C\), go to State 4 if given \(E\), or fail to match the regular expression if given any other character.

Using automata to recognize regular expressions introduces two types of complexity: automata storage and processing costs. A theoretical worst-case study [8] shows that a single regular expression of length \(n\) can be expressed as an NFA with \(O(n)\) states. When the NFA is converted into a DFA, it may generate \(O(\Sigma^n)\) states, where \(\Sigma\) is the set of symbols. The processing complexity for each character in the input is \(O(1)\) in a DFA, but is \(O(n^2)\) for an NFA when all \(n\) states are active at the same time.

To handle \(m\) regular expressions, two choices are possible: processing them individually in \(m\) automata, or compiling them into a single automaton. The former is used in Snort [3] and Linux L7-filter [12]. The latter is proposed in recent studies [5,6] so that the single composite NFA can support shared matching of common prefixes of those expressions. Despite the demonstrated performance gains over using \(m\) separate NFAs, in practice this approach still experiences large numbers of active states. This has the same worst-case complexity as the sum of \(m\) separate NFAs. Therefore, this approach on a serial processor can be slow, as given any input character, each active state must be serially examined to obtain new states.

In DFA-based systems, compiling \(m\) regular expressions into a composite DFA provides guaranteed performance benefit over running \(m\) individual DFAs. Specifically, a composite DFA reduces processing cost from \(O(m)\) (cost of \(O(1)\) for each automaton) to \(O(1)\), i.e., a single lookup to obtain the next state for any given character. However, the number of states in the composite automaton grows to \(O(\Sigma^{mn})\) in the theoretical worst case as listed in Table 10.2.

![Fig. 10.2 An NFA example](image-url)

### Table 10.2 Worst-case comparisons of DFA and NFA

<table>
<thead>
<tr>
<th></th>
<th>One regular expression of length (n)</th>
<th>(m) regular expressions compiled together</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Processing complexity</td>
<td>Storage cost</td>
</tr>
<tr>
<td>NFA</td>
<td>(O(n^2))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>DFA</td>
<td>(O(1))</td>
<td>(O(\Sigma^n))</td>
</tr>
</tbody>
</table>
10.3.3 Lazy DFA

There is a middle ground between DFA and NFA, called lazy DFA. Lazy DFAs are designed to reduce the memory consumption of conventional DFAs [7, 14]: a lazy DFA keeps a subset of the DFA states that match the most common strings in memory; for uncommon strings, it extends the subset from the corresponding NFA at runtime. As such, a lazy DFA is usually much smaller than the corresponding fully compiled DFA and provides good performance for common input strings. The Bro intrusion detection system [1] adopts this approach. However, malicious senders can easily construct packets with uncommon strings to keep the system busy and slow down the matching process. As a result, the system will start dropping packets and malicious packets can sneak through.

10.4 Regular Expression Matching in Network Scanning Applications

The previous section surveyed the most representative traditional approaches to regular expression matching. In this section, we show that these techniques do not always work efficiently for some of the complex patterns that arise in networking applications. To this end, we first enumerate the pattern structures that are common in networking applications in Section 10.4.1. We then show that certain pattern-structures that occur in networking applications are hard to match using traditional methods – they incur either excessive memory usage or high computation cost. In particular, we show two categories of regular expressions in Section 10.4.2 that lead to quadratic and exponential numbers of states, respectively.

10.4.1 Patterns Used in Networking Applications

We study the complexity of DFAs for typical patterns used in real-world packet payload scanning applications such as Linux L7-filter (as of Feb 2006), SNORT (Version 2.4), and Bro (Version 0.8V88). Table 10.3 summarizes the results.\(^1\)

- Explicit strings generate DFAs of size linear in the number of characters in the pattern. Twenty-five percent of the networking patterns, in the three applications we studied (Linux L7-filter, SNORT, and Bro), fall into this category and they generate relatively small DFAs with an average of 24 states.
- If a pattern starts with ‘*’, it creates a DFA of polynomial complexity with respect to the pattern length \(k\) and the length restriction \(j\) on the repetition of a class of

\(^1\) This study is based on the use of exhaustive matching and one-pass search defined in [16].
Table 10.3 An analysis of patterns in network scanning applications

<table>
<thead>
<tr>
<th>Pattern features</th>
<th>Example</th>
<th>Number of states</th>
<th>Percent of patterns</th>
<th>Average # of states</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Case 1</strong>: Explicit strings with ( k ) characters</td>
<td>( ^*ABCD ) ( .^*ABCD )</td>
<td>( k + 1 )</td>
<td>25.1</td>
<td>23.6</td>
</tr>
<tr>
<td><strong>Case 2</strong>: Wildcards</td>
<td>( ^*AB.^*CD ) ( .^*AB.^*CD )</td>
<td>( k + 1 )</td>
<td>18.8</td>
<td>27.2</td>
</tr>
<tr>
<td><strong>Case 3</strong>: Patterns with ( ^* ), a wildcard, and a length restriction ( j )</td>
<td>( ^*AB{j+}CD ) ( ^*AB{0,j})CD</td>
<td>( O(k\times j) )</td>
<td>44.7</td>
<td>180.3</td>
</tr>
<tr>
<td><strong>Case 4</strong>: Patterns with ( ^* ), a class of characters overlaps with the prefix, and a length restriction ( j )</td>
<td>( ^*A+[A-Z]{j})D ( ^*A[A-Z]{j+})D ( )where ( [A-Z] ) overlaps with prefix A+ ( )</td>
<td>( O((k + j^2) \times j \sim 370) )</td>
<td>5.1</td>
<td>136,903</td>
</tr>
<tr>
<td><strong>Case 5</strong>: Patterns with a length a wildcard restriction ( j ), where or a class of characters overlaps with the prefix</td>
<td>( .^*AB.{j})CD ( .^*A[A-Z]{j+})D ( )</td>
<td>( O((k + 2/j) \times j \sim 344) )</td>
<td>6.3</td>
<td>( &gt;2^{344} )</td>
</tr>
</tbody>
</table>

characters in the pattern. Our observation from the existing payload scanning rule sets is that the pattern length \( k \) is usually limited. The length restriction \( j \) is usually small too, unless it is for buffer overflow attempts. In that case, \( j \) will be more than 300 on average and sometimes even reaches thousands. Therefore, Case 4 in Table 10.3 can result in a large DFA because it has a factor quadratic in \( j \). Although this type of pattern only constitutes 5.1% of the total patterns, they create DFAs with an average of 136,903 states.

- There are also a small percent (6.8%) of patterns starting with “\( ^* \)” and having length restrictions (Case 5). These patterns create DFAs of exponential sizes. We will address Cases 4 and 5 in detail in Section 10.4.2.

We compare the regular expressions used in three networking applications, namely, SNORT, Bro, and the Linux L7-filter, against those used in emerging Extensible Markup Language (XML) filtering applications [5, 6] where regular expressions are matched over text documents encoded in XML. The results are shown in Table 10.4. We observe three main differences:

1. While both types of applications use wildcards (‘.’, ‘?’, ‘+’, ‘*’), the patterns for packet-scanning applications contain larger numbers of them. Many such patterns use multiple wildcard metacharacters (e.g., ‘.’, ‘*’). For example, the pattern for identifying the Internet radio protocol, “membername.*session.*player”, has two wildcard fragments “.*”. Some even contain over 10 such wildcard fragments. As regular expressions are converted into state machines for pattern matching, large numbers of wildcards bring multiple matching choices to the matching process, causing the corresponding DFAs to grow exponentially.
Table 10.4  Comparison of regular expressions in networking applications against those in XML filtering

<table>
<thead>
<tr>
<th></th>
<th>SNORT</th>
<th>Bro</th>
<th>L7-filter</th>
<th>XML filtering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of regular expressions analyzed</td>
<td>1555</td>
<td>2780</td>
<td>70</td>
<td>1,000–100,000</td>
</tr>
<tr>
<td>Percent of patterns starting with ‘^’</td>
<td>74.4%</td>
<td>2.6%</td>
<td>72.8%</td>
<td>0.80%</td>
</tr>
<tr>
<td>Percent of patterns with wildcards ‘,.+,?,*’</td>
<td>74.9%</td>
<td>98.8%</td>
<td>75.7%</td>
<td>50%–100%</td>
</tr>
<tr>
<td>Average Number of wildcards per pattern</td>
<td>4.7</td>
<td>4.3</td>
<td>7.0</td>
<td>1–2</td>
</tr>
<tr>
<td>Percent of patterns with class ‘[]’</td>
<td>31.6%</td>
<td>65.8%</td>
<td>52.8%</td>
<td>0</td>
</tr>
<tr>
<td>Average number of classes per pattern</td>
<td>8.0</td>
<td>3.4</td>
<td>4.8</td>
<td>0</td>
</tr>
<tr>
<td>Percent of patterns with length restrictions on classes or wildcards</td>
<td>56.3%</td>
<td>23.8%</td>
<td>21.4%</td>
<td>≈0</td>
</tr>
</tbody>
</table>

(2) Classes of characters (“[“]”) are used in packet-scanning applications, but not in XML processing applications. In addition, the class of characters may intersect with other classes or wildcards. For example, the pattern for detecting buffer overflow attacks to the Network News Transport Protocol (NNTP) is “\(^{SEARCH}\s+[^\n]{1024}\)”, where a class of character “[^\n]” interacts with its preceding white space characters “\s+”. When given an input with SEARCH followed by a series of white spaces, there is ambiguity whether these white spaces match \s+ or the non-return class “[^\n]”. As we will show later in Section 10.4.2.1, such interaction can result in a highly complex state machine.

(3) A high percentage of patterns in packet payload scanning applications have length restrictions on some of the classes or wildcards, while such length restrictions usually do not occur in XML filtering. For example, the pattern for detecting Internet Message Access Protocol (IMAP) email server buffer overflow attack is as follows “.* AUTH \s[^\n]{100}”. This pattern contains the restriction that there would be 100 non-return characters “[^\n]” after matching of keyword AUTH and any number of white spaces “\s”. As we shall show in Section 10.4.2.2, such length restrictions can increase the resource needs for regular expression matching.

10.4.2  Analysis of Regular Expressions That Generates Large DFAs

We mentioned previously that some patterns generate DFAs of quadratic size (Case 4 of Table 10.3) and some others generate exponential-sized DFAs (Case 5 of Table 10.3). Next, we explain these two cases in more detail.

10.4.2.1  DFAs of Quadratic Size

A common misconception is that patterns starting with ‘^’ create simple DFAs. In fact, even in the presence of ‘^’, classes of characters that overlap with the prefix
pattern can still yield a complex DFA. Consider the pattern “B+[^\n]{3}D”, where the class of character [^\n] denotes any character but the return character ‘\n’.

Figure 10.3 shows that the corresponding DFA has a quadratic number of states. The quadratic complexity comes from the fact that the letter B overlaps with the class of character [^\n] and, hence, there is inherent ambiguity in the pattern: the second B letter can be matched either as part of B+, or as part of [^\n]{3}. Therefore, if an input contains multiple B’s, the DFA needs to remember the number of B’s it has seen and their locations i to make a correct decision with the next input character. If the class of characters has length restriction of j bytes, DFA needs \(O(j^2)\) states to remember the combination of distance to the first B and the distance to the last B.

Seventeen patterns in the SNORT rule set fall into this quadratic state category. For example, the regular expression for the NNTP rule is “SEARCH\s+[^\n]{1024}”. Similar to the example in Figure 10.3, \s overlaps with [^\n]. White space characters cause ambiguity of whether they should match \s+ or be counted as part of the 1024 non-return characters [^\n]{1024}. For example, an input of SEARCH followed by 1024 white spaces and then 1024 ‘A’’s will have 1024 ways of matching strings, i.e., one white space matches \s+ and the rest as part of [^\n]{1024}, or two white spaces match \s+ and the rest as part of [^\n]{1024}, and so on. By using 1024^2 states to remember all possible sequences of these white spaces, the DFA accommodates all the ways to match the substrings of different lengths. Note that all these substrings start with SEARCH and hence are overlapping matches.

This type of quadratic state problem cannot be solved by an NFA-based approach. Specifically, the corresponding NFA contains 1042 states; among these, the first six are for the matching of SEARCH, the next one for the matching of \s+, and the rest of the 1024 states for the counting of [^\n]1024 with one state for each count. An intruder can easily construct an input as SEARCH followed by 1024 white spaces. With this input, both the \s+ state and all the 1023 non-return states would
be active at the same time. Given the next character, the NFA needs to check these 1024 states sequentially to compute a new set of active states, hence significantly slowing down the pattern-matching speed.

### 10.4.2.2 DFAs of Exponential Size

In real life, many payload scanning patterns contain an exact distance requirement. Figure 10.4 shows the DFA for an example pattern “.* A..CD”. An exponential number of states \(2^{2^1}\) are needed to represent these two wildcard characters. This is because we need to remember all possible effects of the preceding As as they may yield different results when combined with subsequent inputs. For example, an input AAB is different from ABA because a subsequent input BCD forms a valid pattern with AAB (AABBCD), but not so with ABA (ABABCD). In general, if a pattern matches exactly \(j\) arbitrary characters, \(O(2^j)\) states are needed to handle the requirement that the distance exactly equals \(j\). This result is also reported in [6]. Similar results apply to the case where the class of characters overlaps with the prefix, e.g., “.* A[A–Z]{j}D”.

Similar structures exist in real-world pattern sets. In the intrusion detection system SNORT, 53.8% of the patterns (mostly for detecting buffer overflow attempts) contain a fixed length restriction. Around 80% of the rules start with ‘\^’; hence, they will not cause exponential growth of DFA. The remaining 20% of the patterns do suffer from the state explosion problem. For example, consider the rule for detecting IMAP authentication overflow attempts, which uses the regular expression “.* AUTH\s[\^\n]{100}”. This rule detects any input that contains AUTH, then a white space, and no return character in the following 100 bytes. If we directly compile this pattern into a DFA, the DFA will contain more than 10,000 states because

![Diagram](image)

**Fig. 10.4** A DFA for pattern “.* A..CD” that generates exponential number of states
it needs to remember all the possible consequences that an $AUTH\backslash s$ subsequent to the first $AUTH\backslash s$ can lead to. For example, the second $AUTH\backslash s$ can either match $\backslash \backslash n\{100\}$ or be counted as a new match of the prefix of the regular expression.

It is obvious that the exponential blow-up problem cannot be mitigated by using an NFA-based approach. The NFA for the pattern “.* $AUTH\backslash s[^\backslash n]\{100\}” is shown in Figure 10.5. Because the first state has a self-loop marked with $\Sigma$, the input “$AUTH\backslash sAUTH\backslash sAUTH\backslash s . . .” can cause a large number of states to be simultaneously active, resulting in significantly degraded system performance, as demonstrated in [16].

In the next three sections, we review existing technologies that aim to resolve these problems and speed up regular expression matching in networking applications. In particular, we begin with regular expression rewriting techniques that can significantly reduce DFA sizes (Section 10.5). We next review a DFA compression techniques to further reduce DFA sizes (Section 10.6). Since the fast memory on routers is limited, we also discuss techniques that split DFAs and only keep a compact representation of frequently visited portions in fast memory (Section 10.7).

### 10.5 Regular Expression Rewriting Techniques

Having identified the typical patterns that yield large DFAs, in this section we investigate possible rewriting of some of those patterns to reduce the DFA size. Such rewriting is enabled by relaxing the requirement of exhaustive matching to that of non-overlapping matching (Section 10.5.1). With this relaxation, we propose pattern rewriting techniques that explore the potential of trading off exhaustive pattern matching for memory efficiency for quadratic patterns (Section 10.5.2) and exponential patterns (Section 10.5.3). Finally, we offer guidelines to pattern writers on how to write patterns amenable to efficient implementation (Section 10.5.4).

#### 10.5.1 Rationale Behind Pattern Rewriting

Most existing studies of regular expressions focus on a specific type of evaluation, that is, checking if a fixed length string belongs to the language defined by a regular expression. More specifically, a fixed length string is said to be in the language of a
regular expression, if the string is matched from start to end by a DFA corresponding to that regular expression. In contrast, in packet payload scanning, a regular expression pattern can be matched by the entire input or specific substrings of the input\(^2\). Without a priori knowledge of the starting and ending positions of those substrings (unless the pattern starts with ‘ˆ’ that restricts it to be matched at the beginning of the line, or ends with ‘$’ that limits it to be matched at the end of the line), the DFAs created for recognizing all substring matches can be highly complex. This is because the DFA needs to remember all the possible subprefixes it has encountered. When there are many patterns with a lot of wildcards, they can be simultaneously active (recognizing part of the pattern). Hence, a DFA needs many states to record all possible combinations of partially matched patterns.

For a better understanding of the matching model, we next present a few concepts pertaining to the completeness of matching results and the DFA execution model for substring matching. Given a regular expression pattern and an input string, a complete set of results contains all substrings of the input that the pattern can possibly match. For example, given a pattern “ab*” and an input “abbb”, four possible matches can be reported, a, ab, abb, and abbb. We call this style of matching *Exhaustive Matching*. It is formally defined as below:

**Exhaustive Matching** Consider the matching process \(M\) as a function from a pattern \(P\) and a string \(S\) to a power set of \(S\), such that \(M(P, S) = \{\text{substring } S' \text{ of } S \mid S' \text{ is accepted by the DFA of } P\}\).

In practice, it is expensive and often unnecessary to report all matching substrings, as most applications can be satisfied by a subset of those matches. For example, if we are searching for the Oracle user name buffer overflow attempt, the pattern may be ““USR\(\backslash s[^\n]*\n\{100,\}””, which searches for packets starting with “USR\(\backslash s” and followed by 100 or more non-return characters. An incoming packet with “USR\(\backslash s” followed by 200 non-return characters may have 100 ways of matching the pattern because each combination of the “USR\(\backslash s” with the sequential 100 to 200 characters is a valid match of the pattern. In practice, reporting just one of the matching results is sufficient to detect the buffer overflow attack. Therefore, we propose a new concept, *Non-overlapping Matching*, that relaxes the requirements of exhaustive matching.

**Non-overlapping Matching** Consider the matching process \(M\) as a function from a pattern \(P\) and a string \(S\) to a set of strings, specifically, \(M(P, S) = \{\text{substring } S_i \text{ of } S \mid \forall S_i, S_j \text{ accepted by the DFA of } P, S_i \cap S_j = \emptyset\}\).

If a pattern appears in multiple locations of the input, this matching process reports all non-overlapping substrings that match the pattern. We revisit our example above. For the pattern “ab*” and the input “abbb”, the four matches overlap by sharing

---

\(^2\) The techniques presented in this chapter assume packets are reassembled into a stream before checking for patterns. For pattern matching on out of order packets, please refer to [9].
the prefix \( a \). If we assume non-overlapping matching, we only need to report one match instead of four.

For most payload scanning applications, we expect that non-overlapping matching would suffice, as those applications are mostly interested in knowing if certain attacks or application layer patterns appear in a packet. In fact, most existing scanning tools like grep [2] and flex [13] and systems like SNORT [3] and Bro [1] implement special cases of non-overlapping matches such as left-most longest match or left-most shortest match. As we show later in this section, by restricting the solutions for non-overlapping matches, we can construct more memory-efficient DFAs.

### 10.5.2 Rewrite Rules for DFAs with Quadratic Size

Section 10.4.2.1 showed an example pattern, “ˆSEARCH\s+[\`\n]{1024}”, that can generate DFAs of quadratic size with respect to the length restriction. Below, we explain the intuition behind the pattern rewrite using the above example and then address more general cases.

When identifying non-overlapping patterns, “ˆSEARCH\s+[\`\n]\{1024\}” can be rewritten to “ˆSEARCH\s[\`\n]\{1024\}”. The new pattern specifies that after matching string SEARCH and a single white space \( \s \), it starts counting non-return characters for \[\`\n]\{1024\} regardless of the content. In this way, the ambiguity of matching \( \s \) is removed. It is not hard to see that for every matching substring \( s \) that the original pattern reports, the new pattern produces a substring \( s \) that is either identical to \( s \) or is a prefix of \( s \). In other words, the new pattern essentially implements non-overlapping left-most shortest match. It is also easy to see that the new pattern requires a number of states linear in the length restriction \( j \) (1024 in the example).

This rewrite rule can be applied to a more general case where the suffix of a pattern contains a class of characters overlapping with its prefix and a length restriction, e.g., “\( A+[A-Z]\{j\} \)”. It is proved in [16] that this type of patterns can be rewritten to “\( A[A-Z]\{j\} \)” with equivalence guaranteed under the condition of non-overlap matching. This rewrite rule can also be extended to patterns with various types of length restriction such as “\( A+[A-Z]\{j+1\} \)” and “\( A+[A-Z]\{j,k\} \)”.

### 10.5.3 Rewrite Rule for DFAs of Exponential Size

As discussed in Section 10.4.2.2, patterns like “\( .* AUTH\s[\`\n]\{100\} \)” generate exponential numbers of states to keep track of all the AUTH\s subsequent to the first AUTH\s. If non-overlapping matching is used, the intuition of our rewriting is that after matching the first AUTH\s, we do not need to keep track of the second AUTH\s. This is because:
If there is a ‘\n’ character within the next 100 bytes, the return character must also be within 100 bytes to the second AUTH\.s.

- If there is no ‘\n’ character within the next 100 bytes, the first AUTH\.s and the following characters have already matched the pattern.

The intuition is that we can rewrite the pattern such that it only attempts to capture one match of the prefix pattern. Following the intuition, we can simplify the DFA by removing the states that deal with the successive AUTH\.s. As shown in Figure 10.6, the simplified DFA first searches for AUTH in the first four states, then looks for a white space, and after that starts to count and check whether the next 100 bytes contains a return character. After rewriting, the DFA only contains 106 states.

The rewritten pattern can be derived from the simplified DFA shown in Figure 10.6. We can transform this DFA to an equivalent NFA in Figure 10.7 using standard automaton transform techniques [8]. The transformed NFA can be directly described using the following regular expression:

\[(\[^A\]A[^U]\[^AU]\[^T]\[^AUTH]\[^H\] AUTH[^s\[^\n\]]0,99\[^n\]) * AUTH \[^s\[^\n\]]100\]

This pattern first enumerates all the cases that do not satisfy the pattern and then attaches the original pattern to the end of the new pattern. In other words,
“.*” is replaced with the cases that do not match the pattern, represented by

\[(\[^n\)]A[^U]\)A[^T]\|AUTH[^H]\|AUTH[^s]\|AUTH[^n]\{0, 99\}\n)*

Then, when the DFA comes to the states for AUTH[^n]\{100\}, it must be able to match the pattern. Since the rewritten pattern is directly obtained from a DFA of size \(j + 5\), it generates a DFA of a linear number of states rather than an exponential number before applying the rewrite.

More generally, it is proven in [16] that pattern “.*AB[A-Z]{j}” can be rewritten as “(([^A][^B])AB[A-Z]{j-1}[^A-Z])AB[A-Z]{j}” for detecting non-overlapping strings. Similar rewrite rules apply to patterns in other forms of length restriction, e.g., “.*AB[A-Z]{j+}”.

In [16], these two rewriting rules are applied to the Linux L7-filter, SNORT, and Bro pattern sets. While the Linux L7-filter pattern set does not contain any pattern that needs to be rewritten, the SNORT pattern set contains 71 rules that need to be rewritten and the Bro pattern set contains 49 such patterns (mostly imported from SNORT). For both types of rewrite, the DFA size reduction rate is over 98%.

10.5.4 Guidelines for Pattern Writers

From the analysis, we can see that patterns with length restrictions can sometimes generate large DFAs. In typical packet payload scanning pattern sets including Linux L7-filter, SNORT, and Bro, 21.4–56.3% of the length restrictions are associated with classes of characters. The most common of these are “[^n]”, “[\"]”, (not ‘]’), and “[\"]” (not “”), used for detecting buffer overflow attempts. The length restrictions of these patterns are typically large (233 on the average and reaching up to 1024). For these types of patterns, we highly encourage the pattern writer to add “^” so as to avoid the exponential state growth. For patterns that cannot start with “^”, the pattern writers can use the techniques shown in Section 10.5.3 to generate patterns with linear numbers of states to the length restriction requirements.

Even for patterns starting with “^”, we need to avoid the interactions between a character class and its preceding character, as shown in Section 10.5.2. One may wonder why a pattern writer uses \s+ in the pattern “SEARCH\s+[^n]\{1024\}”, when it can be simplified as \s. Our understanding is that, in reality, a server implementation of a search task usually interprets the input in one of the two ways: either skips a white space after SEARCH and takes the following up to 1024 characters to conduct a search, or skips all white spaces and takes the rest for the search. The original pattern writer may want to catch intrusion into systems of either implementation. However, the way the original pattern is written, it generates false positives if the server does the first type of implementation (skipping all the white spaces). This is because if an input is followed by 1024 white spaces and then some non-whitespace regular command of less than 1024 bytes, the server can skip these white spaces and take the follow-up command successfully. However, this legitimate input will be caught by the original pattern as an intrusion because these white spaces...
themselves can trigger the alarm. To catch attacks to this type of server implementation, while not generating false positives, we need the following pattern:

“\texttt{SEARCH} \backslash s + [^\backslash s][^\backslash n] \{1023\}”

In this pattern, \texttt{\textbackslash s +} matches all white spaces and \texttt{[^\textbackslash s]} means the first non-white space character. If there are more than 1023 non-return characters following the first non-white space character, it is a buffer overflow attack. By adding \texttt{[^\backslash n]}, the ambiguity in the original pattern is removed; given an input, there is only one way to match each packet. As a result, this new pattern generates a DFA of linear size. To generalize, we recommend pattern writers to avoid all the possible overlaps between the neighboring segments in the pattern. Here, overlap denotes an input can match both segments simultaneously, e.g., \texttt{\textbackslash s +} and \texttt{[^\backslash n]}. Overlaps will generate a large number of states in a DFA because the DFA needs to enumerate all the possible ways to match the pattern.

10.6 D\textsuperscript{2}FA: Algorithms to Reduce DFA Space

The pattern rewriting schemes presented in the previous section reduce the DFA storage overhead by reducing the number of DFA states. Besides the states, the DFA storage overhead is also affected by the links between states. This section discusses link compression techniques.

If no link compression is applied, each state in the DFA has $2^8 = 256$ possible outgoing links, one for each ASCII alphabet input. Usually not all outgoing links are distinct. Therefore, table compression techniques can be used to efficiently represent the identical outgoing links \cite{4}. However, these techniques are reported to be inefficient when applied to networking patterns because on the average one state has more than 50 distinct next states \cite{11}.

Kumar et al. proposed Delayed Input DFA (D\textsuperscript{2}FA), a new representation of regular expressions for reducing the DFA storage overhead. Instead of compressing identical links originated from the one state, it compresses links across states based on the observation that multiple states in a DFA can have identical outgoing links. Therefore, linking these states together through default transitions can remove the need of storing outgoing links in each state separately. For a concrete example, consider three states $s_1$, $s_2$, and $s_3$ and their outgoing links in Table 10.5. States $s_1$ and

\begin{table}[h]
\centering
\caption{Three states in a DFA and their transitions for different input characters}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
 & A & B & C & D \\
\hline
State $s_1$ & $s_2$ & $s_3$ & $s_4$ & $s_5$ \\
\hline
State $s_2$ & $s_2$ & $s_7$ & $s_4$ & $s_5$ \\
\hline
State $s_3$ & $s_2$ & $s_7$ & $s_6$ & $s_5$ \\
\hline
\end{tabular}
\end{table}
$s_2$ have identical next states on inputs $A$, $C$, and $D$. Only character $B$ leads to different next states. Similarly, $s_2$ and $s_3$ have identical next states except for character $C$. Instead of these three states storing next states separately, $D^2$FA only stores one outgoing link for $s_2$ (for character $B$), and one for $s_3$ (for character $C$). For other states, $s_2$ can have a default link to $s_1$ and $s_3$ has a default link to $s_2$, where identical links are stored. In this way, the storage overhead of $D^2$FA can be significantly smaller than the original DFA.

To construct $D^2$FA from DFA, one can check the number of identical outgoing links between any two states and use that as a weight function. The weight indicates the number of links that can be eliminated in $D^2$FA. Figure 10.8(a) shows weights in the previous example. The goal of default link selection is to pick default links between states that shared the highest weights. Note that a default path must not contain cycles because otherwise it may bring the $D^2$FA into an infinite loop at some given input. Therefore, the default paths can create a tree or forests. Figure 10.8(b) and (c) are two example selections and (b) has a higher weight than (c). In [11], maximum weight spanning tree algorithms are used to create the default paths and consequently convert DFA to $D^2$FA.

The storage savings of $D^2$FA come at the cost of multiple memory lookups. In the previous example, if using the DFA, given an input $A$ at current state $s_2$, we can obtain the next states with one table lookup. With $D^2$FA, Figure 10.8(b), two memory lookups are necessary. First, we perform one lookup to find out that $s_2$ has no stored outgoing link for character $A$ and we obtain the default link to $s_1$. Next, we perform another lookup into $s_1$ to retrieve the next state for $A$, which is $s_2$. Default links can be connected to form a default path. With a default path, multiple memory lookups are needed. For example, given an input $A$ at state $s_3$ in Figure 10.8(b), we need two extra memory lookups, one following the default link from $s_3$ to $s_2$ and the other from $s_2$ to $s_1$. To generalize, given an input, the number of memory lookups is the number of default links followed plus one. In the worst case, the longest default path becomes the system bottleneck. Therefore, when constructing $D^2$FA from DFA, it is critical to bound the maximum default paths lengths. A spanning tree-based heuristic algorithm is used in [11] for this purpose.

---

**Fig. 10.8** The default link selection example
D$^2$FA is very effective when applied to networking pattern sets. It was able to remove more than 95% of the transitions between states [11], which greatly reduces the memory consumption.

10.7 Bifurcated, History-Augmented DFA Techniques

The previous sections presented rewrite and compression techniques to reduce the storage overhead of DFAs for fast packet pattern processing. DFA sizes can be further reduced by using data structures and algorithms particularly suitable for these packet patterns. In particular, Kumar et al. [10] identified several limitations of traditional DFA-based approaches to packet pattern processing and proposed techniques to overcome these limitations, resulting in compact representations of multiple patterns for high-speed packet content scanning.

A foremost limitation of traditional DFA-based approaches is that they employ complete patterns to parse the packet content. These approaches fail to exploit the fact that normal packet streams rarely match more than the first few symbols of any pattern. As a result, the automata unnecessarily explode in size as they attempt to represent the patterns in their entirety even if the tail portions of the patterns are rarely visited. To overcome this limitation, a key idea in [10] is to isolate frequently visited portions of the patterns, called pattern prefixes, from the infrequent portions, called pattern suffixes. Another important observation of common packet patterns is that the prefixes are generally simpler than the suffixes. Hence, such prefixes can be implemented using a compact DFA representation and stored in a fast memory, expediting the critical path of packet scanning. On the other hand, the suffixes can be implemented using DFAs if they fit in memory, or even using NFAs since they are expected to be executed only infrequently. Such a prefix and suffix-based architecture is referred to as a bifurcated pattern-matching architecture.

There is an important tradeoff in such a bifurcated pattern-matching architecture: On the one hand, we want to make the prefixes small so that the automaton that is active all the time is compact and fast. On the other hand, very small prefixes can be matched frequently by normal data streams, causing frequent invocations of the slow processing of the complex suffixes. Hence, a good solution must strike an effective balance between the two competing goals. The solution proposed in [10] is sketched below with some simplification:

1. Construct an NFA for each packet pattern and execute all those NFAs against typical network traffic. For each NFA, compute the probability with which each state of the NFA becomes active and the probabilities with which the NFA makes its various transitions. The NFAs for two example patterns and their transition probabilities are illustrated in Figure 10.9.

2. Once these probabilities are computed, determine a cut in the NFA graph such that (i) there are as few nodes as possible on the left-hand side of the cut and (ii) the probability that the states on the right-hand side of the cut are active is sufficiently small. Such a cut is illustrated in Figure 10.9(b).
3. After the cut is determined, a composite DFA is constructed for all the prefixes of the NFAs on the left-hand side of the cut. A DFA or NFA is chosen for each suffix on the right-hand side depending on the available memory.

Experimental results in [10] show that more than 50% reduction of memory usage can be achieved for a spectrum of pattern sets used in network intrusion detection systems, when the DFAs for entire patterns are replaced with the DFAs for the prefixes of those patterns obtained using the above technique.

A second limitation of traditional DFA-based approaches is that given a set of patterns to be matched simultaneously with the input data, a composite DFA maintains a single state of execution, which represents the combination of all the partial matches of those patterns. As a result, it needs to employ a large number of states to remember various combinations of the partial matches. In particular, an exponential blowup of the DFA can occur when multiple patterns consist of a simple sequence of characters followed by a Kleene closure [8] over a class of characters, e.g., the two prefixes on the left-hand side of the cut in Figure 10.9(b). In this scenario, the DFA needs to record the power set of the matching results of such prefixes using individual states, hence the exponential size of the machine [10, 16].

To mitigate the combinatorial effect of partial matches of multiple patterns, a history-augmented DFA [10], H-FA, equips the composite DFA with a small auxiliary memory that uses a set of history flags to register the events of partial matches. H-FA further identifies the fragments of the composite DFA that perform similar processing of the input data and only differ in the True/False values of a history flag. In many cases, these DFA fragments can be merged with additional conditions imposed on the DFA transitions and appropriate set and reset operations of the history flag. As reported in [10], such H-FAs can result in more than 80% space reduction for most common pattern sets used in network intrusion detection systems.
10.8 Summary

We considered the implementation of fast regular expression matching for packet-payload scanning applications. Naïve DFA implementations can result in exponentially growing memory costs for some of the patterns used in networking applications. Since reducing this memory cost is critical for any high-speed implementation, we discussed several memory-efficient DFA-based approaches that substantially reduce memory usage. While these techniques do not handle all possible cases of dramatic DFA growth, they do cover those patterns that are present in common payload scanning rule sets like SNORT and Bro. Hence, fast DFA-based pattern matching is feasible for today’s payload-scanning applications.

References