Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection

Fang Yu
Zhifeng Chen
Yanlei Diao
T.V. Lakshman
Randy H. Katz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-76
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-76.html

May 22, 2006

Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection

Fang Yu, *Member, IEEE* Zhifeng Chen, *Member, IEEE* Yanlei Diao, *Member, IEEE*
T. V. Lakshman , *Fellow, IEEE* Randy H. Katz, *Fellow, IEEE*

***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 against a set of patterns specified as regular expressions. In this paper, we first show that memory requirements using traditional methods are prohibitively high for many patterns used in packet scanning applications. We then propose regular expression rewrite techniques that can effectively reduce memory usage. Further, we develop a grouping scheme that can strategically compile a set of regular expressions into several engines, resulting in remarkable improvement of regular expression matching speed without much increase in memory usage. We implement a new DFA-based packet scanner using the above techniques. Our experimental results using real-world traffic and patterns show that our implementation achieves a factor of 12 to 42 performance improvement over a commonly used DFA-based scanner. Compared to the state-of-art NFA-based implementation, our DFA-based packet scanner achieves 50 to 700 times speedup.**

I. Introduction

Packet content scanning (also known as Layer-7 filtering or payload scanning) is crucial to network security and network monitoring applications. In these applications, 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, etc.

Currently, regular expressions are replacing explicit string patterns as the pattern matching language of choice in packet scanning applications. Their widespread use is due to their expressive power and flexibility for describing useful patterns. For example, in the Linux Application Protocol Classifier (L7-filter) [1], all protocol identifiers are expressed as regular expressions. Similarly, the Snort [2] intrusion detection system has evolved from no regular expressions in its ruleset in April 2003 to 1131 out of 4867 rules using regular expressions as of February 2006. Another intrusion detection system, Bro [3], also uses regular expressions as its pattern language.

F. Yu, Department of EECS, University of California Berkeley, Berkeley, CA 94720 (phone: 510-642-8284; email: fyu@eecs.berkeley.edu).
Z. Chen, Google Inc., 1600 Amphitheathre Pkwy, Mountain View, CA 94043 (email: zhifengc@google.com)
Y. Diao, Department of Computer Science, University of Massachusetts Amherst, Amherst, MA 01003 (email: yanlei@cs.umass.edu).
T.V. Lakshman, Bell Laboratories, Lucent Technologies, 101 Crawfords Corner Road, Holmdel, NJ 07733 (email: lakshman@research.bell-labs.com).
Randy H. Katz, Department of EECS, University of California Berkeley, Berkeley, CA 94720 (email: randy@eecs.berkeley.edu).

As regular expressions gain widespread adoption for packet content scanning, it is imperative that regular expression matching over the packet payload keep up with the line-speed packet header processing. Unfortunately, this requirement cannot be met in many existing payload scanning implementations. For example, when all 70 protocol filters are enabled in the Linux L7-filter [1], we found that the system throughput drops to less than 10Mbps, which is well below current LAN speeds. Moreover, over 90% of the CPU time is spent in regular expression matching, leaving little time for other intrusion detection or monitoring functions. On the other hand, although many schemes for fast string matching [4-11] have been developed recently in intrusion detection systems, they focus on explicit string patterns only and can not be easily extended to fast regular expression matching.

The inefficiency in regular expression matching is largely due to the fact that the current solutions are not optimized for the following three unique complex features of regular expressions used in network packet scanning applications.

- First, 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 patterns even contain over ten such wildcard fragments. As regular expressions are converted into state machines for pattern matching, large numbers of wildcards can cause the corresponding *Deterministic Finite Automaton* (DFA) to grow exponentially.
- Second, a majority of the wildcards are used with length restrictions ('?', '+'). As we shall show later in the paper, such length restrictions can increase the resource needs for expression matching.
- Third, groups of characters are also commonly used: for example, the pattern for matching the ftp protocol, "^*220*[\x09-\x0d -~]**ftp*", contains a class (inside the brackets) that includes all the printing characters and space characters. The class of characters may intersect with other classes or wildcards. Such interaction can result in a highly complex state machine.

To the best of our knowledge, there has not been any detailed study of optimizations for these kinds of regular expressions as they are so specific to network packet scanning applications. In this paper, we address this gap by analyzing these regular expressions and developing memory-

efficient DFA-based solutions for high speed processing. Specifically, we make the following contributions:

- We analyze the computational and storage cost of building individual DFAs for matching regular expressions, and identify the structural characteristics of the regular expressions in networking applications that lead to exponential growth of DFAs, as presented in Section 3.2.
- Based on the above analysis, we propose two rewrite rules for specific regular expressions in Section 3.3. The rewritten rules can dramatically reduce the size of resulting DFAs, making them small enough to fit in memory. We prove that the patterns after rewriting are equivalent to the original ones for detecting non-overlapping patterns. While we do not claim to handle all possible cases of dramatic DFA growth (in fact the worse case cannot be improved), our rewrite rules do cover those patterns present in common payload scanning rulesets like Snort and Bro, thus making fast DFA-based pattern matching feasible for today's payload scanning applications.
- We further develop techniques to intelligently combine multiple DFAs into a small number of groups to improve the matching speed in Section IV, while avoiding the exponential growth in the number of states in memory.

We demonstrate the effectiveness of our rewriting and grouping solutions through a detailed performance analysis using real-world payload scanning pattern sets. As the results show, our DFA-based implementation can increase the regular expression matching speed on the order of 50 to 700 times over the NFA-based implementation used in the Linux L7-filter and Snort system. It can also achieve 12-42 times speedup over a commonly used DFA-based parser. The pattern matching speed can achieve gigabit rates for certain pattern sets. This is significant for implementing fast regular expression matching of the packet payload using network processors or general-purpose processors, as the ability to more quickly and efficiently classify enables many new technologies like real-time worm detection, content lookup in overlay networks, fine-grained load balancing, etc.

II. Problem Statement

In this section, we first discuss regular expressions used in packet payload scanning applications, then present the possible solutions for regular expression matching, and finally define the specific problem that we address in this paper.

2.1 Regular Expression Patterns

A regular expression describes a set of strings without enumerating them explicitly. Table 1 lists the common features of regular expression patterns used in packet payload scanning. For example, consider a regular expression from the Linux L7-filter [1] for detecting Yahoo traffic: "^(*ymsg*|*ypns*|*yhoo*).?.?.?.?.?.?.?[*lwt*].**xc0\x80*". This pattern matches any packet payload that starts with *ymsg, ypns,* or *yhoo*, followed by seven or fewer arbitrary characters, and then a letter *l, w* or *t*, and some arbitrary characters, and finally the ASCII letters *c0* and *80* in the hexadecimal form.

Table 2 compares the regular expressions used in two networking applications, Snort and the Linux L7-filter, against those used in emerging Extensible Markup Language (XML) filtering applications [12, 13] where regular expressions are matched over text documents encoded in XML. We notice three main differences: (1) While both types of applications use wildcards ('.', '?', '+', '*'), the patterns for packet scanning applications contain larger numbers of them in each pattern; (2) classes of characters ("[]") are used only in packet scanning applications; (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. This shows that compared to the XML filtering applications, network packet scanning applications face additional challenges These challenges lead to a significant increase in the complexity of regular expression matching, as we shall show later in this paper.

Table 1. Features of Regular Expressions

Syntax	Meaning	Example
^	Pattern to be matched at the start of the input	^AB means the input starts with AB. A pattern without '^', e.g., AB, can be matched anywhere in the input.
|	OR relationship	*A*|*B* denotes *A* or *B*.
.	A single character wildcard	
?	A quantifier denoting one or less	*A*? denotes *A* or an empty string.
*	A quantifier denoting zero or more	*A** means an arbitrary number of *A*s.
{}	Repeat	*A*{100} denotes 100 *A*s.
[ ]	A class of characters	[*lwt*] denotes a letter *l*, *w*, or *t*.
[^]	Anything but	[^\n] denotes any character except \n.

Table 2. Comparison of regular expressions in networking applications against those in XML filtering

	Snort	L7-filter	XML filtering
# of regular expressions analyzed	1555	70	1,000-100,000
% of patterns starting with "^"	74.4%	72.8%	≥80%
% of patterns with wildcards "., +, ?, *"	74.9%	75.7%	50% - 100%
Average # of wildcards per pattern	4.65	7.03	1-2
% of patterns with class "[ ]"	31.6%	52.8%	0
Average # of classes per pattern	7.97	4.78	0
% of patterns with length restrictions on classes or wildcards	56.3%	21.4%	≈0

2.2 Solution Space for Regular Expression Matching

Finite automata are a natural formalism for regular expressions. There are two main categories: *Deterministic Finite Automaton* (DFA) and *Nondeterministic Finite Automaton* (NFA). In this section, we survey existing solutions using these two types of automata.

A DFA consists of a finite set of input symbols, denoted as $\sum$, a finite set of states, and a transition function δ [14]. In networking applications, $\sum$ contains the 2^8 symbols from the extended ASCII code. Among the states, there is a single start state q_0 and a set of accepting states. The transition function δ 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 only one active state in the DFA. An NFA works similarly to a DFA except that the δ function maps from a

state and a symbol to a set of new states. Therefore, multiple states can be active simultaneously in an NFA.

A theoretical worst case study [14] 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 $O(\sum^n)$ states. 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 [2] and Linux L7-filer [1]. The latter is proposed in recent studies [12, 13] 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 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 DFA. Specifically, a composite DFA reduces processing cost from $O(m)$ ($O(1)$ 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 $O(\sum^{mn})$ in the theoretical worst case. In fact, we will show in Section 4 that typical patterns in packet payload scanning applications indeed interact with each other and can cause the creation of an exponential number of states in the composite DFA.

Table 3. Worst case comparisons of DFA and NFA

	One regular expression of length n		m regular expressions compiled together	
	Processing complexity	Storage cost	Processing complexity	Storage cost
NFA	$O(n^2)$	$O(n)$	$O(n^2m)$	$O(nm)$
DFA	$O(1)$	$O(\sum^n)$	$O(1)$	$O(\sum^{nm})$

There is a middle ground between DFA and NFA called lazy DFA. Lazy DFA are designed to reduce memory consumption of conventional DFA [12, 15]: a lazy DFA keeps a subset of the DFA that matches 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. Bro intrusion detection systems [3] adopt this approach. However, malicious senders can easily construct packets that keep the system busy and slow down the matching process.

Field Programmable Gate Arrays (FPGAs) provide a high degree of parallelism and thus can be used to speed up the regular expression matching process. There are existing FPGA solutions that build circuits based on DFA [16] or NFA [17-19]. These approaches are promising if the extra FPGA hardware can be embedded in the packet processors. FPGAs, however, are not available in many applications; in such situations, a network processor or general-purpose CPU-based implementation may be more desirable.

2.3 Problem statement

In this paper, we seek a fast and memory-efficient solution to regular expression matching for packet payload scanning. We define the scope of the problem as follows:

- We consider ***DFA-based approaches*** in this paper, as NFA-based approaches are inefficient on serial processors or processors with limited parallelism (e.g., multi-core CPUs in comparison to FPGAs). Our goal is to achieve $O(1)$ computation cost for each incoming character, which cannot be accomplished by any existing DFA-based solutions due to their excessive memory usage. Thus, the focus of the study is to reduce memory overhead of DFA while approaching the optimal processing speed of $O(1)$ per character.
- We focus on ***general-purpose processor-based architectures*** and explore the limits of regular expression matching in this environment. Wherever appropriate, we leverage the trend of multi-core processors that are becoming prevalent in those architectures. Nevertheless, our results can be used in FPGA-based and ASIC-based approaches as well [20].

It is worth noting that there are two sources of memory usage in DFAs: states and transitions. The number of transitions is linear with respect to the number of states because for each state there can be at most 2^8 (for all ASCII characters) links to next states. Therefore, we consider the number of states (in minimized DFA) as the primary factor for determining the memory usage in the rest of the paper. Also, due to the need for high performance, we do not consider DFAs that use any table compression techniques.

III. Matching Of Individual Patterns

In this section, we present our solution to matching individual regular expression patterns. The main technical challenge is to create DFAs that can fit in memory, thus making a fast DFA-based approach feasible. We first define a few concepts key to DFA construction in the context of packet payload scanning in Section 3.1. We then analyze the size of DFAs for typical payload scanning patterns in Section 3.2. Although theoretical analyses [12, 14] have shown that DFAs are subject to exponential blow-up, here, we identify *specific* structures that can lead to exponential growth of DFAs. Based on the insights from this analysis, in Section 3.3, we propose pattern rewrite techniques that explore the possibility of trading off exhaustive pattern matching (which real-world applications often allow) for memory efficiency. Finally, we offer guidelines to pattern writers on how to write patterns amenable to efficient implementation in Section 3.4.

3.1 Design Considerations

Although regular expressions and automata theory can be directly applied to packet payload scanning, there is a noticeable difference in between. Most existing studies on regular expressions focus on a specific type of evaluation, that is, checking if a fixed length string belongs to the language that a regular expression defines. 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. Without a priori knowledge of the starting and ending positions of those substrings, DFAs created for recognizing all substring matches can be highly complex.

For a better understanding, we next present a few concepts pertaining to the completeness of matching results and the DFA execution model for substring matching.

Completeness of matching results

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*, three possible matches can be reported, *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, $M(P, S)$ = {substring S' of S| S' 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. 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, $M(P, S)$ = {substring S_i of S| $\forall\ S_i, S_j$ accepted by the DFA of P, $S_i \cap S_j = \phi$ }.

If a pattern appears in multiple locations of the input, this matching process reports all non-overlapping substrings that match the pattern. Revisit our example above. For the pattern *ab** and the input *abbb*, the three matches overlap by sharing the prefix *ab*. For this example, non-overlapping matching will report one match instead of three.

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 and flex and systems like Snort [2] and Bro [3] implement special cases of non-overlapping matching such as left-most longest matching or left-most shortest matching. As we shall show later this section, non-overlapping matching can be exploited to construct more memory-efficient DFAs.

DFA execution model for substring matching

In the following discussion, we focus on patterns without '^' attached at the beginning. Recall that for such patterns, there is no prior knowledge of whether/where a matching substring may appear. To handle these patterns, two types of DFAs can be created with different execution models:

Repeated searches. A DFA can be created directly from a pattern using standard DFA construction techniques [14]. To find the set of matching substrings (using either exhaustive or non-overlapping matching), the DFA execution needs to be augmented with repeated searches of the input: An initial search starts from the beginning of the input, reading characters until (1) it has reported all matches (if exhaustive matching is used) or one match (if non-overlapping matching is used), or (2) it has reached the end of the input. In the former case, the new search will start from the next character in input (if exhaustive matching is used) or from the character after the reported match (if non-overlapping matching is used). In the latter case, a new search is initiated from the next character in input. This style of repeated scanning using DFA is commonly used in language parsers. However, it is inefficient for packet payload scanning where the chance of the packet payload matching a particular pattern is low (such inefficiency is verified in Section 5.3.3).

One-pass search. In the second approach, ".*" is pre-pended to each pattern without '^', which explicitly states that the pattern can be matched anywhere in the input. Then a DFA is created for the extended pattern. As the input is scanned from start to end, the DFA can recognize all substring matches that may start at different positions of the input. Using one pass search, this approach can truly achieve $O(1)$ computation cost per character, thus suitable for networking applications. To achieve high scanning rate, we adopt this approach in the rest of the study.

3.2 DFA Analysis for Individual Regular Expressions

Next, we study the complexity of DFA for typical patterns used in real-world packet payload scanning applications such as Linux L7-filter, Snort, and Bro. The study is based on the use of *exhaustive matching* and *one-pass search*. Table 4 summarizes the results.

- Explicit strings generate DFAs of length linear to the number of characters in the pattern.
- If a pattern starts with '^', it creates a DFA of polynomial complexity with respect to the pattern length k and the length restriction j. Our observation from the existing payload scanning rule sets is that the pattern length k is usually limited but the length restriction j can reach hundreds or even thousands. Therefore, Case 4 can result in a large DFA because it has a factor quadratic in j.
- Patterns starting with ".*" and having length restrictions (Case 5) cause the creation of DFA of exponential size.

Table 4. Analysis of patterns with *k* characters

Pattern features	Example	# of states
1. Explicit strings with k characters	^*ABCD* .**ABCD*	$k+1$
2. Wildcards	^*AB.*CD* .**AB.*CD*	$k+1$
3. Patterns with ^, a wildcard, and a length restriction j	^*AB.*{j+}*CD* ^*AB.*{0, j}*CD* ^*AB.*{j}*CD*	$O(k*j)$
4. Patterns with ^, a class of characters overlaps with the prefix, and a length restriction j	^*A*+[*A-Z*]{j}D	$O(k+j^2)$
5. Patterns with a length restriction j, where a wildcard or a class of characters overlaps with the prefix	.**AB.*{j}*CD* .**A*[*A-Z*]{j+}D	$O(k+2^j)$

Next, we explain the two cases of large DFA sizes, namely, Case 4 and Case 5 of Table 4, in more detail.

Case 4: DFA of Quadratic Size

A common misconception is that patterns starting with '^' create simple DFAs. However, we discover that even with

'^', 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). Its corresponding DFA has a quadratic number of states, as shown in Figure 1. 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: A 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 in order 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*.

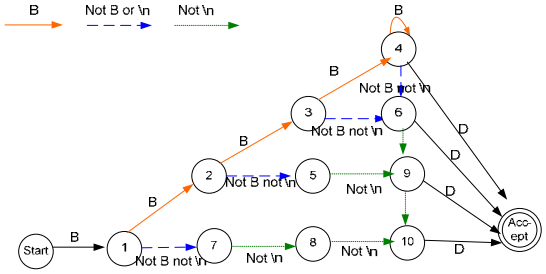


Figure 1. A DFA for Pattern *^B+[^\n]{3}D*

Similar structures in real world pattern sets:

A significant number of patterns in the Snort rule set fall into this category. For example, the regular expression for the NNTP rule is "*^SEARCH\s+[^\n]{1024}*". Similar to the example in Figure 1, *\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}*. Specifically, 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}*, etc. By using 1024^2 states to remember all possible consequences 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, one is for the matching of *SEARCH*, 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.

This problem cannot be solved by a fixed string pre-filtering scheme (used by Snort), either. This is because pre-filtering can only recognize the presence of the fixed string "*SEARCH*" in the input. After that, an NFA or DFA-based matching scheme is still needed in post processing to report whether the input matches the pattern and what the matches are. Another choice is to count the subsequent characters in post processing after identifying the prefix "*SEARCH*". This approach does not solve the problem because every packet (even normal traffic) with the prefix will incur the counting process. In addition, intruders can easily construct packets with multiple (different) prefixes to invoke many requests for such post processing.

Case 5: DFA of Exponential Size

Many payload scanning patterns contain an ***exact distance*** requirement. Figure 2 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 *A*s 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 *exact j* requirement. This result is also reported in [12]. Similar results apply to the case where the class of characters overlaps with the prefix, e.g., "*.*A[A-Z]{j}D*".

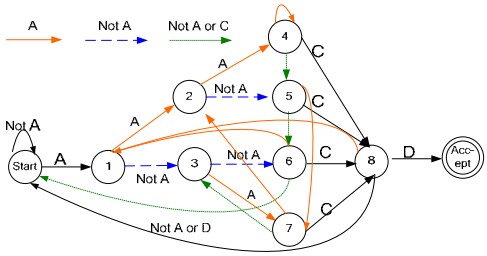


Figure 2. A DFA for pattern *.*A.{2}CD*

Similar structures 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. Out of them, 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 rule into a DFA, the DFA will contain more than 10,000 states because it needs to remember all the possible consequences that an *AUTH\s* subsequent to the first *AUTH\s* can lead to. For example, the second *AUTH\s* can either match *[^\n]{100}* or be counted as a new match of the prefix of the regular expression.

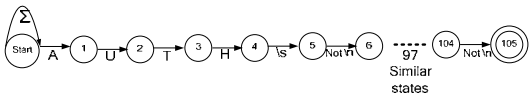
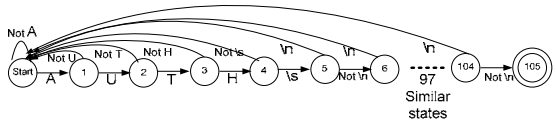


Figure 3. NFA for the pattern *.*AUTH\s[^\n]{100}*

It is obvious that the exponential blow-up problem cannot be mitigated by using an NFA-based approach. The NFA for the pattern "*.*AUTH\s[^\n]{100}*" is shown in Figure 3. Because the first state has a self-loop marked with Σ, the input "*AUTH\sAUTH\sAUTH\s…*" can cause a large number of states to be simultaneously active, resulting in significantly degraded system performance, as demonstrated

by our results reported in Section 5.3.3. Similar to Case 4, this problem cannot be solved by a fixed string pre-filtering scheme (used by Snort), either.



FigureT 4. DFA for rewriting the pattern .AUTH\s[^\n]{100}***

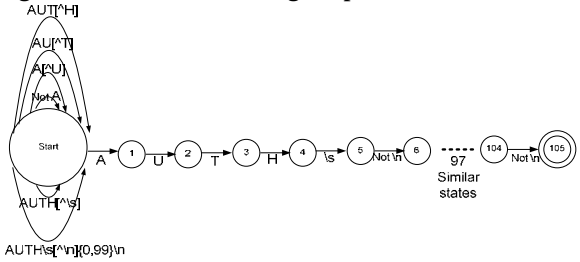


Figure 5. Transformed NFA for deriving Rewrite Rule (1)

3.3 Regular Expression Rewrites

We have identified the typical patterns used in packet payload scanning that can cause the creation of large DFAs. In this section, we investigate the possibility of rewriting 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*. In particular, we propose two rewrite rules, one for rewriting specific patterns belonging to the case of quadratic-sized DFAs (Case 4 in Section 3.2), and the other for rewriting specific patterns that generate exponential-sized DFAs (Case 5 of Section 3.2). The commonality of the patterns amenable to rewrites is that their suffixes address length restricted occurrences of a class of characters that overlap with their prefixes. These patterns are typical in real-world rulesets such as Snort and Bro. For these patterns, as shown in Section 3.2, neither the NFA-based solution nor the fixed string pre-filtering scheme can handle them efficiently. In contrast, our rewrites rules can convert these patterns into DFAs with their sizes successfully reduced from quadratic or exponential to only linear.

Rewrite Rule (1)

As shown in Section 3.2, patterns that start with '^' and contain classes of characters with length restrictions, e.g., "^*SEARCH*\s+[^\n]{*1024*}", can generate DFAs of quadratic size with respect to the length restriction. Below, we first explain the intuition behind Rewrite Rule (1) using the above example and then state a theorem for more general cases.

Given the fact that such patterns are used in packet scanning applications for detecting buffer overflow attempts, it seems reasonable to assume that non-overlapping matches are sufficient for reporting such attacks. Based on this observation, we propose to rewrite the pattern "^*SEARCH*\s+[^\n]{*1024*}" to "^*SEARCH*\s[^\n]{*1024*}". The new pattern specifies that after matching a single white space, we start counting for [^\n]{*1024*} no matter what the content is. 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 *j* because it has removed the ambiguity for matching \s.

We provide a theorem (Theorem 1 in the Appendix) for a more general case where the suffix of a pattern contains a class of characters overlapping with its prefix and a length restriction, "^*A*+[*A-Z*]{*j*}". We prove that this type of pattern can be rewritten to "^*A*[*A-Z*]{*j*}" with equivalence guaranteed under the condition of non-overlap matching. Note that our rewrite rule can also be extended to patterns with various types of length restriction such as "^*A*+[*A-Z*]{*j*+}" and "^*A*+[*A-Z*]{*j*,*k*}". Details are omitted in the interest of space.

Using Rewrite Rule (1), we successfully rewrote 17 similar patterns in the Snort rule set. Detailed results regarding these rewrites are reported in Section 5.2.

Rewrite Rule (2)

As we discussed in Section 3.2, patterns like ".***AUTH*\s[^\n]{*100*}" generate exponential numbers of states to keep track of all the *AUTH*\s subsequent to the fist *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 (1) 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, and (2) if there is no '\n' character within the next 100 bytes, the first *AUTH*\s and the following characters have already matched the pattern. This intuition implies 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 4, the simplified DFA first searches for *AUTH* in the first 4 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.

We derive our rewrite pattern from the simplified DFA shown in Figure 4. Applying a standard technique that maps a DFA/NFA to a regular expression [14], we transform this DFA to an equivalent NFA in Figure 5. For the link that moves from state 1 back to the start state in Figure 4 (i.e., matching *A* then not *U*), the transformed NFA places it right at the start state and labels it with *A*[^*U*]. The transformed NFA does the same for each link moving from state *i* ($1 \le i \le 105$) to the start state in Figure 4. The transformed NFA can be directly described using the following regular expression:

"([^*A*]|*A*[^*U*]|*AU*[^*T*]|*AUT*[^*H*]|*AUTH*[^\s]|*AUTH*\s[^\n]{*0,99*}\n)***AUTH*\s[^\n]{*100*}".

This rule 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 ([^*A*]|*A*[^*U*]|*AU*[^*T*]|*AUT*[^*H*]|*AUTH*[^\s]|*AUTH*\s[^\n]{*0,99*}\n)*. Then, when the DFA comes to the states for *AUTH*\s[^\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 as opposed to an exponential number before the rewrite.

We also provide a theorem (Theorem 2 in the Appendix) that proves the equivalence of the new pattern and the original pattern for a more general case "*.*AB[A-Z]{j}*" under the condition of non-overlapping matching. Moreover, we offer rewrite rules for patterns in other forms of length restriction, e.g., "*.*AB[A-Z]{j+}*".

Rewrite Rule (2) is applicable to 54 expressions in the Snort rule sets and 49 in the Bro rule set. We wrote a script to automatically rewrite these patterns and observed significant reduction in DFA size. Detailed simulation results are reported in Section 5.2.

3.4 Notes for Pattern Writers

As mentioned above, an important outcome of this work is that our pattern rewriter can automatically perform both types of rewriting. An additional benefit is that our analysis provides insight into how to write regular expression patterns amenable to efficient DFA implementation. We discuss this in more detail below.

From the analysis in Section 3.2, we can see that patterns with length restrictions can generate large DFAs. By studying typical packet payload scanning pattern sets including Linux L7-filter, Snort, and Bro, we found that 21.4-56.3% of the length restrictions are associated with classes of characters. The most common classes of characters are "[^\n]", "[^\]]" (i.e., not ']'), and "[^\]", 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 as we showed in Section 3.3. For patterns that cannot start with "^", the pattern writers can use the Rewrite Rule 2 to generate state efficient patterns.

Even for patterns starting with "^", we need to carefully avoid the interactions between a class of characters and its preceding character as shown in Rewrite Rule 1. 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 skip a white space after *SEARCH* and take the following up to 1024 characters to conduct a search, or skip all white spaces and take the rest for the search. The original pattern writer may want to catch intrusion to systems of either implementation. However, the original pattern will generate 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 input will be caught by the original pattern as 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.

"^*SEARCH*\s+[^\s][^\n]{*1023*}"

In this pattern, \s+ matches all white spaces and [^\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 [^\s], the ambiguity in the original pattern is removed; given an input, there is only way of matching each packet. As a result, this new pattern generates a DFA of linear size.

IV. Selective Grouping of Multiple Patterns

The previous section presented our analysis of the complexity of the DFA created for individual patterns and two rewrite techniques that simplify these DFA so that they could fit in memory. As we mentioned in Section 2.2, it is well known that the computation complexity for processing m patterns reduces from $O(m)$ to $O(1)$ per character, when the m patterns are compiled into a single composite DFA. However, it is usually infeasible to compile a large set of patterns together due to the complicated interactions between patterns. In some cases, the composite DFA may experience exponential growth in size, although none of the individual DFA has an exponential component.

In this section, we first present two examples illustrating the interactions between patterns, and then use a real-world payload scanning ruleset to demonstrate the existence of exponential growth in reality. Based on these observations, we propose grouping algorithms that selectively divide patterns into groups while avoiding the adverse interaction among patterns.

4.1 Interactions of Regular Expressions

When patterns share prefixes, some states can be merged. For example, states 1 and 2 shown in Figure 6 are shared by "*.*ABCD"* and "*.*ABAB"*. Combining these patterns can save both storage and computation.

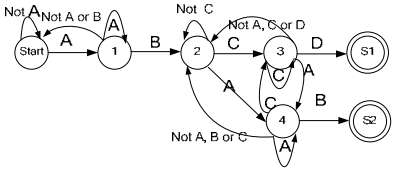


Figure 6. A DFA for pattern *.*ABCD* and *.*ABAB*

However, if the patterns do not share the same prefix, putting m patterns together may generate 2^m states.

Figure 7 shows a composite DFA for matching "*.*AB.*CD"* and "*.*EF.*GH"*. This DFA contains many states that did not exist in the individual DFAs. Among them, state 8 is created to record the case of matching both prefixes *AB* and *EF*. Generally speaking, if there are l patterns with one wildcard per pattern, we need $O(2^l)$ states to record the matching of the power set of the prefixes. In such scenarios, adding one more pattern into the DFA doubles its size. If there are x wildcards per pattern, then $(x+1)^l$ states are required. There are several such patterns in the Linux L7-filter. For example, the pattern for the remote desktop protocol is "*.*rdpdr.*cliprdr.*rdpsnd*", and the pattern for Internet radio is "*.*membername.*session.*player*". Snort also has similar patterns and the number of ".*" in a pattern can go up to six.

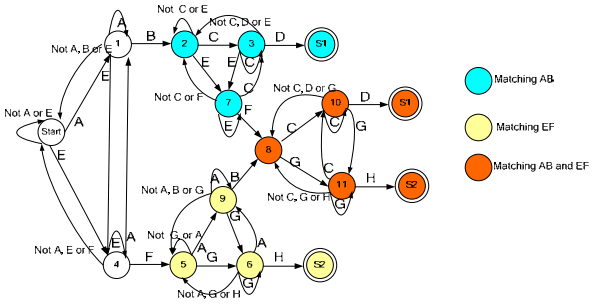


Figure 7. A DFA for pattern *.*AB.*CD* and *.*EF.*GH*

4.2 Interactions of Real-world Regular Expressions

We study the pattern interactions of the Linux L7-filter [1] in this section. If 70 patterns are compiled separately into 70 DFAs, each DFA has tens to hundreds of states. The total number of states is 3533. When we start to group multiple patterns into a composite DFA (we select patterns with a random order), the processing complexity decreases. However, the total number of DFA states (i.e., the sum of the composite DFA and those ungroup ones) grows over 136,786 with just 40 patterns, as illustrated by the increasing dotted line in Figure 8. We could not add more patterns into the composite DFA because it exceeded the memory limit in the test machine that we used (1.5 GB). However, not all patterns cause significant DFA growth. Only some patterns (e.g., pattern 12, 37, and 38 as shown in Figure 8) lead to significant growth of the DFA. These patterns all contain large numbers of wildcards, and sometimes have classes of characters. For example, pattern 12 contains a fixed length (20) of wildcards, pattern 37 contains three unrestricted wildcards (".*"), and pattern 38 contains 19 classes of characters, 4 unrestricted wildcards, and 8 length restricted wildcards.

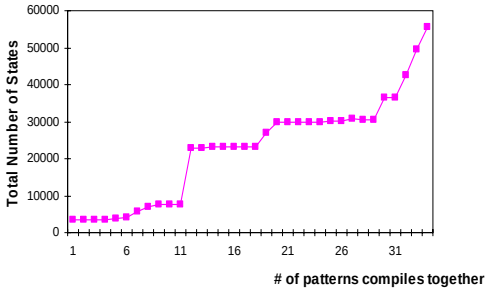


Figure 8. DFA Size and Processing Complexity of Multiple Patterns (Unsorted order)

4.3 Regular Expressions Grouping Algorithms

As discussed above, certain patterns interact with each other when compiled together, which can result in a large composite DFA. In this section, we propose algorithms to selectively partition *m* patterns to *k* groups such that patterns in each group do not adversely interact with each other. As such, these algorithms reduce the computation complexity from $O(m)$ to $O(k)$ without causing extra memory usage.

We first provide a formal definition of *interaction*: two patterns interact with each other if their composite DFA contains more states than the sum of two individual ones. To calculate the number of states in the composite DFA, we first construct an NFA by adding a new start state, two ε edges leading to the individual DFA, a new accepting state, and two ε edges from the DFA accepting states to the new accepting state, as shown in Figure 9. Then we run the NFA to DFA conversion algorithm and the DFA minimization algorithm to obtain the composite DFA.

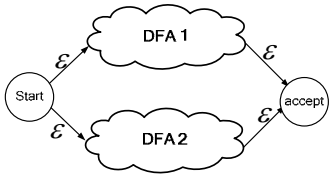


Figure 9. Composite NFA for two DFAs

We use the information on pairwise interaction to group a set of *m* regular expressions. The intuition is that if there is no interaction between any pair selected from R_1, R_2, and R_3, the composite DFA of R_1, R_2, R_3 is not likely to exceed the sum of individual ones. We validate this point using empirical results in Section 5.3.1.

We devise grouping algorithms both for multi-core processor architecture, where groups of patterns can be processed in parallel among different processing units, and for general processor architecture, where the DFA for one group corresponds to one process or thread. Next, we present the algorithm for the former architecture first and then the algorithm for the latter.

In multi-core architecture, there are multiple parallel processing units. Their number is usually limited, e.g., 16 in Intel IXP2800 NPU, which is much smaller than the number of patterns. Hence, one DFA per pattern per processing unit is infeasible. Our goal is to design an algorithm that divides regular expressions into several groups, so that one processing unit can run one or several composite DFAs. In addition, the size of local memory of each processing unit is quite limited. For example, the newly architected IBM cell processor has 8 synergistic processor elements, each with 128KB local memory [23]. Hence, we need to keep grouping patterns until they meet the local memory limit. The pseudo-code of the algorithm is provided below.

In this algorithm, we first compute the pairwise interaction of regular expressions. With this pairwise information, we construct a graph with each pattern as a vertex and an edge between patterns R_i and R_j if they interact with each other. Using this graph, we can start with a pattern that has least interaction with others, and then try to add patterns that have least interactions into the same group. We keep adding until the composite DFA is larger than the local memory limit. Then we proceed to create a new group from the patterns that remain ungrouped.

For regular expression R_i in the set
  For regular expression R_j in the set
    Compute pairwise interaction of R_i and R_j

Construct a graph $G(V, E)$
  V is the set of regular expressions, with one vertex per regular expression
  E is the set of edges between vertices, with an edge (V_i, V_j) if R_i and R_j interact with each other.

Repeat
  New group (NG) = ϕ

Pick a regular expression that has the least interaction with others and add it into new group *NG*
Repeat
Pick a regular expression *R* has the least number of edges connected to the new group
Compile $NG \cup \{R\}$ into a DFA
if this DFA is larger than the limit
break;
else
Add *R* into *NG*
Until every regular expression in *G* is examined
Delete *NG* from *G*
Until no regular expression is left in *G*

Algorithm for Multi-core Processor Architecture with limited total memory size

General processor architecture. In the general processor architecture, if there are multiple composite DFAs to be run, the processor executes each of them sequentially. Usually all the DFAs are kept in the main memory for the performance purpose. Since the memory is shared among all DFAs, we want to group all patterns into the smallest number of groups (hence the smallest number of DFA) while not exceeding the available memory size. It is clear that finding the smallest number of groups is an NP hard problem. In this work, we apply heuristics to find a small number of groups that can serve as a good approximation. The pseudo-code of our algorithm for the general processor architecture is shown in the following.

Leftover memory *L* = Total memory
For regular expression R_i in the set
Compute the DFA size D_i for R_i
Leftover memory -= D_i

Repeat
New group (NG) = ϕ
Pick a regular expression which has the least interaction with others and add it into new group NG
Repeat
Pick a regular expression *R* that has the least number of edges connected to the new group
Compile $NG \cup \{R\}$ into a DFA
if $D(NG) > \sum_{R_i \in NG} D(R_i) + L*|NG|/(\#\text{left patterns})$
break;
else
Add *R* into *NG*
Until every regular expression in *G* is examined
Leftover memory L -= $D(NG) - \sum_{R_i \in NG} D(R_i)$
Delete *NG* from *G*
Until no regular expression is left in *G*

Algorithm for General-Processor Architecture

Different from the multi-core case, in this algorithm we first compute the DFA of individual patterns and compute the leftover memory size. At any stage, we always try to distribute the leftover memory evenly among the ungrouped expressions, which is the heuristics that we apply to increase the number of grouping operations, hence reducing the number of resulting groups. In this algorithm, we group patterns using a similar routine as the previous algorithm. However, we stop grouping when the size of the composite DFA (denoted as $D(NG)$) exceeds its share of the leftover memory. Here, the DFA's share of the leftover memory is calculated using the formula = (Leftover memory *L*) * (Number of patterns in the group) / (Number of ungrouped patterns).

Discussion: Grouping multiple regular expressions into one composite DFA is a well known technique to enhance matching speed. Our algorithms focus on picking the right patterns to be grouped together. Similar to our approach, systems like Bro group patterns into one group, instead of several groups. They adopt a lazy DFA-based approach, where they cache commonly used DFA states and extend the DFA at run-time if needed. The distinction between our approach and Bro's approach is that our grouping algorithm produces scanners of deterministic complexity. The lazy DFA-based approach, although fast and memory efficient on most common inputs, may be exploited by intruders to construct malicious packets that force the lazy DFA to enter many corner cases [15]. Our fully-developed DFA does not have performance degradation under such attacks.

V. EVALUTION RESULTS

We implement a DFA scanner with rewriting and grouping functionality for efficient regular expression matching. In this section, we evaluate the effectiveness of our rewriting techniques for reducing DFA size, and the effectiveness of our grouping algorithms for creating memory-efficient composite DFA. We also compare the speed of our scanner against a DFA-based repeated scanner generated by flex [25] and a best-known NFA-based scanner [26]. Compared to the DFA-based repeated scanning approach, our DFA-based one pass scanning approach has 12 to 42 times performance improvements. Compared to the NFA-based implementation, our DFA scanner is 50 to 700 times faster on traffic dumps obtained form MIT and Berkeley networks.

5.1 Experimental Setup

To focus on regular expressions commonly used in networking applications, we select the following three complex pattern sets: The first is from the Linux layer 7 filter [1] which contains 70 regular expressions for detecting different protocols. The second is from the Snort system [2] which contains 1555 regular expressions for intrusion detection. The last one is from Bro intrusion detection system [3] with a total of 2781 regular expressions.

We use two sets of real-world packet traces. The first set is the intrusion detection evaluation data set from the MIT DARPA project [24]. It contains more than a million packets. The second data set is from a local LAN with 20 machines at the UC Berkeley networking group, which contains more than six million packets. The characteristics of MIT dump are very different from Berkeley dump. MIT dump mostly contains intrusion packets that are long, with the average packet payload length being 507.386 bytes. In the Berkeley dump, however, most packets are normal traffic, with 67.65

bytes on average in the packet payload. A high percentage of the packets are ICMP and ARP packets that are very short.

We use Flex [25] to convert regular expressions into DFAs. Our implementation of the DFA scanner eliminates backtracking operations [25]. It only performs one-pass search over the input and is able to report matching results at the position of the end of each matching substring.

All the experimental results reported were obtained on PCs with 3.4 Ghz CPU and 3.7 GB memory.

5.2 Effect of Rule Rewriting

We apply our rewriting scheme presented in Section 3.2 to the Linux L7-filter, Snort and Bro pattern sets. For the Linux L7-filter pattern set, we do not identify any pattern that needs to be rewritten. For the Snort pattern set, however, 71 rules need to be rewritten. For Bro, 49 patterns (mostly imported from Snort) need to be rewritten using Rewrite Rule 2. For these patterns, we gain significant memory savings as shown in Table 5. For both types of rewrite, the DFA size reduction rate is over 98%.

Table 5. Rewriting effects

Type of Rewrite	Rule Set	Number of Patterns	Average length restriction	DFA Reduction Rate
Rewrite Rule 1: (Quadratic case)	Snort	17	370	>98%
	Bro	0	0	0
Rewrite Rule 2: (Exponential Case)	Snort	54	344	>99%[1]
	Bro	49	214.4	>99%[1]

17 patterns belong to the category for which Rewrite Rule 1 can be applied. These patterns (e.g., "^*SEARCH*\s+[^\n]{*1024*}") all contain a character (e.g., \s) that is allowed to appear multiple times before a class of characters (e.g., [^\n]) with a fixed length restriction (e.g., 1024). As discussed in Section 3.2, this type of pattern generates DFAs that expand quadratically in the length restriction. After rewriting, the DFA sizes come down to linear in the length restriction. A total of 103 patterns need to be rewritten using Rewrite Rule 2. Before rewriting, most of them generate exponential sized DFAs that cannot even be compiled successfully. With our rewriting techniques, the collection of DFAs created for all the patterns in the Snort system can fit into 95MB memory, which can be satisfied in most PC-based systems.

5.3 Effect of Grouping Multiple Patterns

In this section, we apply the grouping techniques to regular expression sets. We show that our grouping techniques can intelligently group patterns to boost system throughput, while avoiding extensive memory usages. We test on three pattern sets: the Linux L7-filter, the Bro http-related pattern set and the Bro payload related pattern set. The patterns of L7-filter can be grouped because the payload of an incoming packet is compared against all the patterns, regardless of the packet header information. For the Bro pattern set, as most rules are related to packets with specific header information, we pick the http related patterns (a total of 648) that share the same header information, as well as 222 payload scanning patterns that share the same header information. Note that we do not report the results of using the Snort rule set because its patterns overlap significantly with those of the Bro rule set.

5.3.1 Interaction of Patterns

For all three pattern sets, a majority of patterns are non-interactive, particularly in Bro http patterns set where all patterns are non-interactive. As a result, most patterns in these rulesets can be combined pair-wise. This nice property offers a significant potential for our grouping algorithms to produce small numbers of groups. To achieve that, one assumption that our grouping algorithms use needs to be verified. As stated in Section 4.2, the assumption is that if three patterns are pair-wise non-interactive, it is highly likely that the size of the composite DFA will not exceed the sum of the individual sizes. We verify this assumption with the real world pattern sets. Table 6 shows that this assumption is valid for over 99.8% of the cases from all three pattern sets.

Table 6. Interaction of regular expressions

	No-interaction Pair-wise	Pair-wise No-interaction lead to No-interaction three patterns
Linux L7-filter	71.18%	99.87%
Bro http	100%	100%
Bro payload	93.3%	99.99%

5.3.2 Grouping Results

We apply our grouping algorithms to all three pattern sets and successfully group all of them into small (<5) numbers of groups. For the Bro's http pattern set, since patterns do not interact with each other, it is possible to compile all 648 patterns into one composite DFA of 6218 states. The other two sets, however, cannot be grouped into one group due to interactions. Below, we report results obtained using our grouping algorithm for the multi-core architecture in Table 7, where local memory is limited. The results for the general processor architecture are in Table 8.

Table 7. Results of grouping algorithms for the multi-core architecture

7 (a) Linux L7-filter (70 Patterns)

Composite DFA state limit	Groups	Total Number of States	Compilation Time (s)
617	10	4267	3.3
2000	5	6181	12.6
4000	4	9307	29.1
16000	3	29986	54.5

7(b) Payload patterns from Bro (222 Patterns)

Composite DFA state limit	Groups	Total Number of States	Compilation Time (s)
540	11	4868	20
1000	7	4656	118
2000	5	5430	780
6000	4	9197	1038

Table 7(a) shows the results for Linux L7-filter pattern set. We start by limiting the number of states in each composite DFA to 617, the size of the largest DFA created for a single pattern in the Linux L7-filter set. The actual memory cost is 617 times 256 next state pointers times *log*(617) bits for each pointer, which amounts to 192 KB.

[1] Note, we use over 99% because some of the patterns create too many states to be compiled successfully without rewriting. 99% is obtained by calculating those successful ones.

Considering that most modern processors have large data caches (>0.5MB), this memory cost for a single composite DFA is comparatively small. Our algorithm generates 10 groups when the limit on the DFA size is set to 617. It creates fewer groups when the limit is increased to larger numbers. As today's multi-core network processors have 8-16 engines, it is feasible to allocate each composite DFA to one processor and take advantage of parallelism.

With our grouping algorithms, we can decrease the number of pattern groups from 70 (originally ungrouped) to 3 groups. This means that, given a character, the generated packet content scanner needs to perform only three state transitions instead of the 70 state transitions that were necessary with the original ungrouped case. This results in a significant performance enhancement (show in Section 5.3.3).

For Bro's payload pattern set, we can group more patterns into one group. As Table 7(b) shows, starting from 540, the largest individual DFA size, the grouping algorithm can group 222 patterns into 11 groups. As the DFA state limit increases, the number of groups decreases down to 4.

Table 8. Results of grouping algorithms for general processor architecture

8(a) Linux L7-filter (70 Patterns)

Total DFA state limit	Groups	Total Number of States	Compilation Time (s)
3533	12	3371	5.602
4000	10	3753	7.335
10000	5	7280	37.928
32000	3	25215	49.976

8(b) Payload patterns from Bro (223 Patterns)

Composite DFA state limit	Groups	Total Number of States	Compilation Time (s)
5221	6	4697	1050
8000	4	6854	1030

Table 8 demonstrates that the grouping algorithm for the general processor architecture can effectively reduce the number of groups generated as the memory limit imposed on the algorithm is increased. In addition, the total number of DFA states is close to the memory limit, showing the algorithm can fully utilize the memory allocated to the packet scanner. Note that we start memory limit at 3533 DFA states for Linux L7-filter which is the total number of the states of individual DFAs. Our simulation results show that we can group 70 patterns into 12 groups with no extra memory usage. Similarly, we start 5221 DFA states for Bro payload set, which is the sum of 233 individual DFAs. Even without extra memory, we can decrease the number of groups from 233 to 6.

Beyond the effectiveness, Table 7 and Table 8 also present the running time of our grouping algorithms. This overhead is a one-time cost. In networking environments, the packet content scanner operates continuously until there are new patterns to be inserted to the system. As patterns in the Linux L7-filter or Bro system do not change frequently, the occasional overhead of several minutes is affordable. In addition, we do not need to regroup all patterns given any new pattern. We can just compute its pairwise interactions with existing patterns and pick a group that yields least total interactions. This type of incremental update computation time is in average less than 1 second on the Bro payload pattern set.

5.3.3 Speed Comparison

We compare our DFA-based algorithms with the starte-of-the-art NFA-based regular expression matching algorithm. Both L7-filter and Snort systems use this NFA-based library. We also compare it with the DFA-based repeated scan approach generated by flex [25]. The results are summarized in Table 9. Our DFA-based one pass scanner is 47.9 to 704 times faster than the NFA-based scanner. Compared to DFA-based repeated scan engine, our scanner yields a performance improvement of 1244% to 4238%. Also note that although these dumps have dramatically different characteristics, our scanner provides similar throughputs over these dumps because it scans each character only once. The other two approaches are subject to dramatic change in throughput (1.8 to 3.4 times) over these traces, because they need to do backtracking or repeated scans. Of course, we admit that the memory usage of our scanner is 2.6 to 8.4 times the NFA-based approach. However, the largest scanner we created (Linux L7-filter, 3 groups) uses 13.3MB memory, which is well under the memory limit of most modern systems.

Table 9. Comparison of the Different Scanners

		Throughputs (Mb/s)		Memory Consumption (KB)
		MIT dump	Berkeley dump	
Linux L-7	NFA	0.98	3.4	1636
	DFA RP	16.3	34.6	7632
	DFA OP 3 groups	690.8	728.3	13596
Bro Http	NFA	30.4	56.1	1632
	DFA RP	117.2	83.2	1624
	DFA OP 1 group	1458	1612.8	4264
Bro Payload	NFA	5.8	14.8	1632
	DFA RP	17.1	25.6	7628
	DFA OP 4 groups	566.1	568.3	4312

NFA—NFA-based implementation
DFA RP – Flex generated DFA-based repeated scan engine
DFA OP – Our DFA one pass scanning engine

VI. Conclusion and Future Work

We considered the implementation of fast regular expression matching for packet payload scanning applications. While NFA-based approaches are usually adopted for implementation because naïve DFA implementations can have exponentially growing memory costs, we showed that with our rewriting techniques, memory-efficient DFA-based approaches are possible. In addition, we presented a scheme that selectively groups patterns together to further speed up the matching process. Our DFA-based implementation is 2 to 3 orders of magnitude faster than the widely used NFA implementation and 1 to 2 orders of magnitude faster than a commonly used DFA-based parser. Our grouping scheme

can be applied to general processor architecture where the DFA for one group corresponds to one process or thread, as well as to multi-core architecture where groups of patterns can be processed in parallel among different processing units. In the future, it would be an interesting study to apply different DFA compression techniques and explore tradeoffs between the overhead of compression and the savings in memory usage.

REFEENCES

[1] J. Levandoski, E. Sommer, and M. Strait, "Application Layer Packet Classifier for Linux." http://l7-filter.sourceforge.net/.

[2] "SNORT Network Intrusion Detection System." http://www.snort.org.

[3] "Bro Intrusion Detection System." http://bro-ids.org/Overview.html.

[4] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," Proc. LISA, 2005.

[5] Y. Cho and W. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," Proc. FCCM, 2004.

[6] Z. K. Baker and V. K. Prasanna, "Time and area efficient pattern matching on FPGAs," Proc. FPGAs, 2004.

[7] Z. K. Baker and V. K. Prasanna, "A methodology for synthesis of efficient intrusion detection systems on FPGAs.," Proc. FCCM, 2004.

[8] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speedup up intrusion detection," Proc. WASSA, 2004.

[9] S. Dharmapurikar, M. Attig, and J. Lockwood, "Deep packet inspection using parallel bloom filters," IEEE Micro, 2004.

[10] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern Matching with TCAM," Proc. ICNP, 2004.

[11] Y. H. Cho and W. H. MangioneSmith, "A Pattern Matching Coprocessor for Network Security," Proc. DAC, 2005.

[12] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML Streams with Deterministic Automata and Stream Indexes," ACM TODS, vol. 29, 2004.

[13] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, "Path Sharing and Predicate Evaluation for High-Performance XML Filtering," ACM TODS, 2003.

[14] J. E. Hopcroft, R. Motwani, and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Second ed: Addison Wesley, 2001.

[15] R. Sommer and V. Paxson, "Enhancing Byte-Level Network Intrusion Detection Signatures with Context," Proc. CCS, 2003.

[16] J. Moscola, J. Lockwood, R. P. Loui, and Michael Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," Proc. FCCM, 2003.

[17] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," Proc. FCCM, 2001.

[18] R. Franklin, D. Carver, and B. Hutchings, "Assisting network intrusion detection with reconfigurable hardware," Proc. FCCM, 2002.

[19] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," Proc FCCM, 2004.

[20] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, J. Turner., "Algorithms to accelerate Multiple Regular Expression Matching for Deep Packet Inspection," Under submission.

[21] "Standard for Information Technology, Portable Operating System Interface (POSIX)," Portable Applications Standards Committee of IEEE Computer Society and the Open Group.

[22] C. L. A. Clarke and G. V. Cormack, "On the use of regular expressions for searching text," Technical Report CS-95-07, Department of Computer Science, University of Waterloo, 1995.

[23] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," IBM J. RES. & DEV., vol. 49, JULY/SEPTEMBER 2005.

[24] "MIT DARPA Intrusion Detection Data Sets." http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html.

[25] V. Paxson et al., "Flex: A fast scanner generator." http://www.gnu.org/software/flex/.

[26] Perl compatible Regular Expression, http://www.pcre.org/

Appendix

Theorem 1: Pattern *P1 "^A[A-Z]{j}"* is equivalent to the original pattern *P2 "^A+[A-Z]{j}"* for detecting non-overlapping shortest string.

(1) Any input matching *P1* must match *P2* as well, and the shortest matched string *S1* for *P1* is the same as the shortest matched sting *S2* for *P2*.

Proof: For any input matching *P1*, it must match pattern *P2* because we can use *"\s"* to match *"\s+"* and the remaining *j* characters as *[A-Z]{j}*. Next we prove their matched string *S1* and *S2* are identical. For *P1*, there is only one way of selecting *S1* and its length is *j+1*. There maybe multiple ways of selecting *S2* (same start position, overlapping strings), with length from *j+1* to infinity. If we pick the shortest match, its length would also be *j+1*. In addition, *S1* and *S2* must start from the beginning of the input due to the requirement of ^. Given that they have the same length, *S1* and S2 must be identical.

(2) Any input matching *P2* must match *P1* as well, and both patterns report matching of the same shortest string.

Proof: For any input matching *P2 "^A+[A-Z]{j}"*, it must have *x "A"s (x>=1)* matched as *"^A+"*, *y "A"s* and *z [A-Z]* characters (starting from a none *"A")* matched as *"[A-Z]{j}" (y>=0, z>=0, y+z=j).* This input must match *P1 "^A[A-Z]{j}"* because the input have *x-1+y+z>=j [A-Z]* characters after the first *A*. Similar to (1), the shortest matched strings are the same.

Since pattern starts with ^, *P1* and *P2* report at most one match for one line. Given (1) and (2), *P1* and *P2* report the same results for any input, hence they are equivalent. □

Theorem 2. Patter *P1* ".*AB[A-Z]{j}" can be rewritten as pattern *P2 "([^A]|A[^B]|AB[A-Z]{j-1}[^(A-Z)])*AB[A-Z]{j}"*. These two patterns are equivalent for detecting non-overlapping strings.

Proof: It is trivial that these two patterns are equivalent when the input does not contain *AB\s* because none of them match the input. It is also trivial if the input only contains one *AB\s*. Next, we prove the case where we have multiple *ABs* without *[^(A-Z)]* in between and they are within *j* bytes to the first *AB* through (1), (2) and (3).

(1) Any input not matching *P2* does not match *P1* either.

Proof: Since the input does not match *P2*, there must be a *[^(A-Z)]* character within the next *j* bytes of the first *AB*, this character must also be located within *j* bytes to the following *ABs*. Hence, *P1* will not be matched either.

(2) Any input matching *P2* must match *P1*. *P2* and *P1* generate matching results at the same position.

Proof: For any input matching *P2*, it must report matching result at *j* positions after the first *AB*. If there is no *[^(A-Z)]* character within the next *j* bytes to one of the *ABs*, then there will not be *[^(A-Z)]* within the next *j* bytes to the first *AB* because there is no *[^(A-Z)]* in between of these *ABs*. Therefore, the match result of *P1* will be generated *j* bytes after the first *AB* as well. Hence, *S1* and *S2* are the same.

(3) *P1* and *P2* report the same number of matches.

Proof: When there are multiple *ABs* without *[^(A-Z)]* between them and they are within *j* bytes to the first *AB*. *P1*

would only report one match, because these *ABs* are within *j* bytes and their matching strings overlap with each other. *P2* would report one match too. Hence, *P1* and *P2* report the same number of matches. If there are multiple non-overlapping patterns in the input (*ABs* are at least *j* apart or with *[^(A-Z)]* in between), *P1* and *P2* still report the same number of matches because we can divide the input to segments, where only one match is reported in one segment.

Given (1), (2) and (3), for any input, patterns P1 and P2 report the same matching results and hence they are equivalent. □