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-2005-8
http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-8.html

October 27, 2005

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

Fang Yu[1] Zhifeng Chen[2] Yanlei Diao[3] T. V. Lakshman[4] Randy H. Katz[1]

[1] University of California Berkeley [2] Google Inc.

[3] University of Massachusetts Amherst [4] Bell Laboratories, Lucent Technologies

Abstract

Packet content scanning at a 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 for fast packet scanning are prohibitively high for many patterns used in networking applications. We then propose regular expression rewrite techniques that reduce memory usage. Further, we develop a scheme based on compiling regular expressions into several engines, which dramatically increases the regular expression matching speed without significantly increasing memory usage. We implement the DFA-based packet scanners. Our experiment results using real-world traffic and patterns have shown that our implementation achieves 3.1 – 4.1 times higher throughput compared to the ungrouped DFA implementation. Compared to the best NFA-based implementation, our DFA-based packet scanner achieves 28-1192 times speedup.

1 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 applications, viruses, protocol definitions, etc.

Currently, regular expressions are replacing explicit string patterns as the pattern matching language of choice in the above networking applications. The wide use of regular expressions is due to their expressive power and flexibility to describe patterns useful in networking. 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 now. Another intrusion detection system, Bro [3], also uses regular expressions as its pattern language.

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 recently developed in intrusion detection systems, they focus on explicit string patterns only and cannot be easily extended to fast regular expression matching.

The inefficiency in regular expression matching largely results from the fact that the current solutions are not highly optimized for the following three unique complex features of regular expressions used in networking applications.

- First, many such patterns use multiple wildcard *metacharacters* (e.g., '.', '*'). For example, the pattern for identifying the Internet radio protocol is "*membername.*session.*player*", which 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 Automata* (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, it is also common to have groups of characters: 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

networking applications. In this paper, we address this gap by analyzing the regular expressions used in networking applications and developing DFA-based solutions for high speed processing. Specifically, we make the following contributions:

- We provide a thorough analysis of the computational and storage cost of building individual DFA for matching regular expressions, and identify the structural characteristics of the regular expressions that lead to exponential growth of DFA, as presented in Section 4.1.
- Based on the above analysis, we propose two rewrite rules for specific regular expressions in Section 4.2, which can dramatically reduce the size of their DFA. As our results using a real-world case study show, these rules play a crucial role in simplifying the DFA so that they could fit in memory, thus making fast DFA-based regular expression matching feasible.
- We further develop techniques to combine multiple DFA into a small number of groups to improve the matching speed. We propose solutions in Section 5 that selectively group patterns into several automata without causing excessive memory usage. These solutions are designed for both general-purpose CPU and multi-core processor architectures.

We demonstrate the effectiveness of our rewriting and grouping solutions through a detailed performance analysis using real-world payload scanning rule sets. As the results show, our DFA-based implementation can increase the regular expression matching speed 28 to 1192 times over the NFA-based implementation used in the Linux L7-filter and Snort system. The pattern matching speed can achieve gigabit rates. 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 such as real-time worm detection, content lookup in overlay networks, fine-grained load balancing etc.

2 Regular Expressions Used in Packet Payload Scanning

A regular expression describes a set of strings without enumerating them explicitly. Let's look at an example regular expression in the Linux L7-filter [1] for detecting Yahoo traffic: "^(*ymsg*|*ypns*|*yhoo*).?.?.?.?.?.?.? [*lwt*].*\xc0\x80". This Yahoo pattern matches any packet payload that starts with *ymsg, ypns,* or *yhoo* ('|' denotes *or* relationship), followed by seven or fewer arbitrary characters ('.' is a single character wildcard and '?' is a quantifier representing one or less), and then a letter *l, w* or *t* (characters inside the brackets form a class), and some arbitrary characters of any length ('*' represents zero or more), and finally the ASCII letters *C0* and *80* in the hexadecimal form. Note that the '^' symbol at the beginning of the pattern requires it to be matched at the start of the packet payload. Patterns without '^', however, can be matched anywhere in the packet payload.

Table 1 compares the regular expressions used in two networking applications, Snort and the Linux L7-filter, against those used in 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 application use wildcards ('.', '?', '+', '*'), the patterns for networking applications contain larger numbers of them in each pattern; (2) Classes of characters ("[]") are used only in networking applications and not in XML filtering; (3) a high percentage of patterns in networking have length restrictions on some of the classes or wildcards, while such length restrictions usually do not occur in XML filtering. This shows that as compared to the XML filtering applications, networking applications typically face an additional challenge in packet classification. Due to these three issues, network applications have more much complexity in regular expression matching.

Table 1. 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

3 Background and Related Work

Finite automata are used to match regular expressions. There are two main categories: *Deterministic Finite Automata* (DFA) and *Nondeterministic Finite Automata* (NFA). In this section, we compare approaches using these two types of finite automaton in the context of networking applications.

A DFA consists of a finite set of input symbols, denoted as Σ, a finite set of states, and a transition function δ [14]. In networking applications, Σ 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. Given an input string, a DFA works as follows: starting from the start state, for each input character, it advances to a new state by following the δ function. If an accepting state is reached, it reports a match of the string. At any time, there is only one active state in a 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 $O(2^n)$ 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 available. One is to compile them into a single automaton and the other is to process them individually in m automata. For using NFA, a recent study [12, 13] proposes to compile m regular expressions into a single NFA with the ability to share the matching of the common prefixes of these expressions, and has demonstrated the performance benefits of this approach over using m separate NFA. Optimized NFA-based approaches, however, can still experience large numbers of active states in practice [12, 13] and have the worst case complexity as the sum of m separate NFA. Consequently, NFA-based approaches on a serial processor can be slow, because given an input character, each active state must be serially examined to obtain new states.

Table 2. 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	$O(1)$	$O(2^n)$	$O(1)$	$O(2^{nm})$

In DFA-based systems, compiling m regular expressions into a composite DFA provides guaranteed performance improvement over running m individual DFAs. Specifically, a composite DFA reduces processing cost from $O(m)$ ($O(1)$ per individual 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 $O(2^{mn})$ in the theoretical worst case. A recent study shows that for simple deterministic patterns, DFA-based approaches exhibit similar memory demands as NFA-based approaches [15]. For regular expressions, however, this result may no longer hold. In fact, we will show in Section 4 that patterns in networking applications interact with each other and can indeed generate an exponential number of states.

There is a middle ground between DFA and NFA called lazy DFA. Since a DFA-based approach may create too many states to fit into memory, lazy DFAs are designed to reduce memory consumption [12, 16]. The main idea is that 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 excellent performance for common input strings. However, malicious senders can easily construct packets that keep the system busy expanding, thus slowing 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 [15] 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 is more desirable. In this paper, we focus on a processor-based architecture and explore the limits of regular expression matching in this environment.

As we discussed before, NFA-based approaches are not effective on serial processors or processors with limited parallelism (as in the case of multi-core CPUs in comparison to FPGAs). Hence, in the rest of the paper, we consider DFA-based approaches. Since a DFA has a fixed $O(1)$ computation cost for each incoming character, we focus on memory cost. There are two sources of memory usage: 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 as the primary factor determining the memory usage in this paper. Finally note that all our studies are based on minimized DFA. Also, due to the need for high performance, we only use DFA that have no table compression and do not need backtracking.

4 DFA Sizes for Single Patterns

In this section, we analyze the size of DFA for typical patterns used in networking applications and identify the types of regular expressions that can lead to exponential growth of DFA. Based on the insights obtained from this analysis, we propose rewrite rules that transform some of those regular expressions to

equivalent regular expressions; the DFA created for the new regular expressions have a significant reduced machine size. As we shall show in Section 6.2, these rewrite rules are highly effective for real-world payload scanning rulesets, and in fact, without these rules, a DFA-based approach for fast regular expression matching could be infeasible in systems with modest memory size.

4.1 DFA State Analysis for Individual Regular Expressions

In this section, we analyze the complexity of a DFA created for a single regular expression. We use patterns in the Linux L7-filter and Snort systems as examples to assist our study. Table 3 summarizes the results.

Explicit strings. Nine out of seventy protocol patterns in the Linux L7-filter can be written explicitly. For example, the pattern for the AOL instant messenger is "*user-agent:aim/*". The DFA of a simple explicit pattern like this uses the same number of states as the length of the pattern, with each state denoting the matching of a character. Links between the states represent transition functions. Figure 1 shows the DFA for patterns *^AACD* and *AACD* (note that these patterns are simpler than the L7-filter patterns but we use them here for ease of exposition). The difference between these two patterns is that *^AACD* must be matched at the beginning of an input but *AACD* can appear anywhere in an input. In Figure 1, the DFA of *^AACD* only includes the solid links between the states. For *AACD*, we include the dashed links to the previous states. For example, at state 3 (where *AAC* have already been matched), another input *A* cannot satisfy the pattern *AACD*, but can cause the transition back to state 1, as we have just matched an *A* and should be waiting for *ACD* to appear.

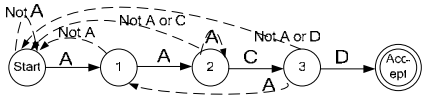

Figure 1. A DFA for pattern *^ABCD* (solid links) and *ABCD* (all links)

Patterns with wildcards. "**.*"** accommodates arbitrary characters of any length. An example is "*cs .*dl.www.counter-strike.net*" which is used to detect packets of a popular online game called Counter Strike. ".*" can be implemented by a single link in a DFA; no extra state is needed. For example, Figure 2 is the DFA for the pattern *AB.*CD*. State 2 denotes the matching of *AB*. From there, if we encounter an input that is not *CD*, we can simply loop at this state awaiting *CD* to appear. In other words, this state now serves as the start state for the pattern *CD*.

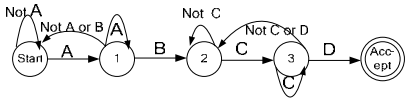

Figure 2. A DFA for pattern *AB.*CD*

Wildcards with length restrictions. Many regular expressions in networking applications impose length restrictions on the wildcard '.' in one of the three forms: equal to, shorter than ('?'), or longer than ('+') than a specific length. For example, the pattern for identifying remote login "*^[a-z][a-z0-9][a-z0-9]+/[1-9][0-9]?[0-9]?[0-9]?00*" requires the matching of two or more alphanumeric letters ([*a-z0-9*][*a-z0-9*]+) after the first letter ([*a-z*]). Later, it looks for three or less digits from [*0-9*]. In the following, we use shorter patterns for ease of illustration. We consider a single wildcard with a length restriction in each pattern. We refer to the pattern fragment before the wildcard as the *prefix pattern* and that after as the *suffix pattern*.

Let us first consider length restrictions with the semantics of ***at least***. Figure 3 shows the DFA for the pattern *AB.{2+}CD*, which requires at least two characters to appear between *AB* and *CD*. Consider the pattern as an equivalent to *AB...*CD*. States 2 and 3 in the DFA attempt to match two arbitrary characters "..", and the rest of the DFA is the same as that for *AB.*CD*. In general, if a pattern requires at least *j* arbitrary characters, we need *j* states to accept these characters.

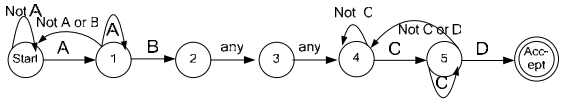

Figure 3. A DFA for pattern *AB.{2+}CD*

The DFA for the ***at most*** case are more complex than those for *at least*. We illustrate this using an example pattern *AB.{0,2}CD*. Its DFA is shown in Figure 4. For the *at least* case, as long as *AB* has been matched, any subsequent *AB* becomes uninteresting to us. For the *at most* case, however, we need to record the matching of a second *AB* because we want to keep track of the distance between *AB* and *CD*. Suppose the input is *ABABGCD*. If we do not keep track of the second *AB* and take it as arbitrary characters, we would miss a match that consists of the second *AB* and the following *CD* (note that *CD* does not form a match with the first *AB* because it is three characters away). Hence, states 3 and 4 are needed for recording the matching of a second prefix *AB*. In addition, extra states are needed to help match pattern *CD*. Suppose there is a *C* right after *AB*. There is ambiguity whether it matches the character *C* or '.'. Therefore, a new state (state 5) is created to capture the case. If a *D* is followed directly, we can go directly to the accepting state. If another *C* follows this *C*, then this *C* can be counted as '.'. In this example, two extra states are needed for matching *AB* and two extra states are needed for *CD*.

In general, if a pattern allows up to *j* arbitrary characters between the prefix (*p* characters) and suffix (*q* characters), then *(p-1)*j* states are needed for matching prefixes, and *(q-1)*j* states are needed for the suffix. A total of *O(j*k)* extra states are needed, where *k* is the pattern length *p+q*. Note that if the pattern starts with '^', the extra states for the prefix are no longer needed because matching a second prefix after the first position is no longer valid. The extra states for the suffix, however, are still required.

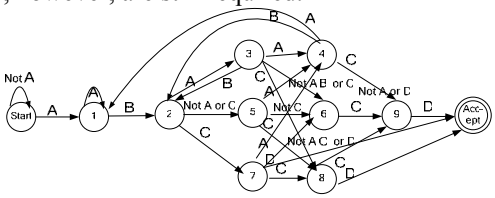

Figure 4. A DFA for pattern *AB.{0,2} CD*

An ***exact distance*** requirement further complicates the structure of DFA. For example, the pattern for the XBOX gaming traffic "^\x58\x80….\xf3|^\x06\x58\x4e" accommodates exactly four arbitrary characters. Figure 5 shows the DFA for a simpler pattern *A..CD*. An exponential number of states (2^{2+1}) are needed to capture these two arbitrary characters. This is because we need to remember all the *A*s in the previous three characters, as different combinations of *A*s may yield different results for the same successive characters. For example, an input *AAB* is different from *ABA* because a successive 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]. All these extra states are not needed when the prefix is required to appear at the beginning of the input using '^'.

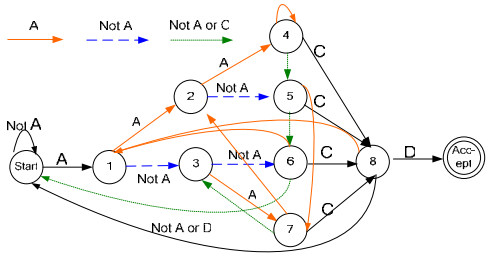

Figure 5. A DFA for pattern *A.{2}CD*

Patterns containing classes of characters. Classes of characters are another source of high complexity. We first examine the patterns without '^'. If a class of the characters (e.g., [*0-9*]) does not overlap with the prefix of the pattern, we can treat them largely as a single character. For example, *A*[*0-9*][*0-9*]*D* has the same number of states as *ABCD*, just with more transition links between two neighboring states, one for each character from the class. However, if the class overlaps with the prefix, e.g., *A*[*A-Z*][*A-Z*]*D*, it would have the same number of states as *A.{2}D*. So it is clear that in the absence of '^', classes of characters that overlap with the prefix can cause exponential growth of DFA.

The presence of '^' eliminates the exponential blowup from the DFA handling classes of characters, as it does for length restrictions. However, even with '^', classes of characters that overlap with the prefix pattern can still create complex DFA. Consider the pattern ^*B*+[^\n]{*3*}, 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 6. The reason is that the letter *B* overlaps with the class of character [^\n]. Hence, there is inherit ambiguity in the pattern: A second *B* letter can be matched either as part of *B*+, or as part of [^\n]. Therefore, if an input contains multiple *B*s, the DFA needs to remember the number of *B*s it has seen and also the distance from the first *B* in order to make a correct decision for 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*. The number of states can be phenomenal for a large *j*.

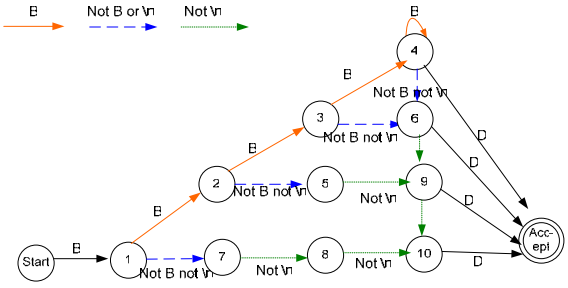

Figure 6. A DFA for Pattern ^*B*+[^\n]{*3*}*D*

Table 3. Analysis of patterns with *k* characters

Pattern features	Example	# of states
1) Explicit strings	^*ABCD* *ABCD*	*k+1*
2) Wildcards	*AB.*CD* ^*AB.*CD*	*k+1*
3) *At least j* arbitrary characters	*AB.{j+}CD* ^*AB.{j+}CD*	*k+j+1*
4) *At most j* arbitrary characters	*AB.{0, j}CD*	*O(k*j)*
5) *At most j* arbitrary characters, ^, and a suffix of *q* characters	^*AB.{0, j}CD*	*O(q*j)*
6) *Exactly j* arbitrary characters	*AB.{j}CD*	$O(k+2^j)$
7) *Exactly j* arbitrary characters and ^	^*AB.{j}CD*	*k+j+1*
8) A class of characters overlapping with the prefix, and a length restriction *j*	*A*[*A-Z*]{*j*}*D*	$O(k+2^j)$
9) A class of characters overlapping with the prefix, a length restriction *j*, and ^	^*A*+[*A-Z*]{*j*}*D*	$O(k+j^2)$

Summary. Table 3 summarizes our analysis. Although it doesn't cover all the possible regular expressions, it shows the DFA size for typical patterns

used in networking applications. It can be clearly seen that case 6 (i.e., exactly *j* arbitrary characters) and case 8 (i.e., a class of characters overlapping with the prefix and with a length restriction *j*) contain an exponential number of states. Other cases have polynomial complexity with the 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 thousands. Based on this, we note that case 9 (i.e., case 8 plus '^') can also result in large DFA as it has a quadratic factor with respect to *j*.

4.2 Rewriting Regular Expressions

Having identified the patterns that can cause large numbers of states to be created in DFA (i.e., cases 6, 8, and 9 in Table 3), we next investigate the possibility of rewriting some of those patterns to reduce the DFA size. The exponential blowup in case 6 is inherent in the problem of allowing exactly *j* arbitrary characters, as we show in our previous analysis. The large DFA in cases 8 and 9, however, result from the operational ambiguity that is caused by the overlap between a class of characters and the prefix pattern. In this section, we present two rewrite rules for these two cases respectively, which transform the original patterns to equivalent ones with the ambiguity eliminated. Consequently, the DFA created for the transformed patterns have a significantly reduced size.

Rewrite rule (1)

55.7% of regular expressions in Snort systems contain classes of characters with length restrictions, mostly for detecting buffer overflow attempts. Out of them, 80% start with "^" and hence are not subject to the exponential state growth problem. The remaining 20%, however, experience exponential blowup in DFA. For a concrete 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, but no return character in the following 100 bytes. If we directly compile this rule into a DFA, the DFA contains more than 10,000 states, because it needs to remember all the possible consequences that the *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 start of regular expression *AUTH*\s. This is similar to the example in Figure 5.

It is obvious that the exponential blowup problem cannot be mitigated by using an NFA-based approach such as L7-filter and Snort systems do. It is because the pattern can cause a huge number of states to be simultaneously active, resulting in severely degraded system performance, as demonstrated by our results reported in Section 6.3.3.

We propose to rewrite the pattern to remove the exponential component of the DFA. The intuition of rewriting is that after matching the first *AUTH*\s, we do not need to care about 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. The new pattern may create fewer matches than the original pattern. For example, if an input starts with "*AUTH AUTH*" and does not contain a return character in the following 105 bytes, the original regular expression can report two matches. But this rewriting is safe because capturing one such case is enough to report buffer overflow attempts.

Following the intuition of finding the first match, we can simplify the DFA by removing the states that remember the successive *AUTH*\s, as shown in Figure 7. 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.

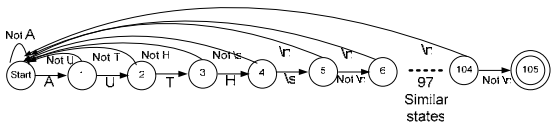

Figure 7. DFA for rewriting the pattern *AUTH*\s[^\n]{*100*}

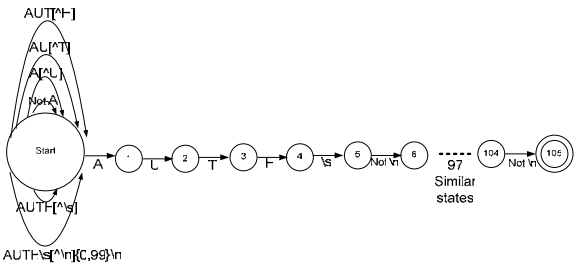

Figure 8. Transformed NFA for deriving rewrite rule (1)

We derive our rewrite rule from this simplified DFA. Applying a standard technique that maps a DFA/NFA to a regular expression [14], we transform this DFA to an equivalent NFA in Figure 8. For the link that moves from state 1 back to the start state in Figure 7 (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 7. 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}".

In summary, we propose a rewrite rule for patterns such as "*AUTH*\s[^\n]{*100*}": This rule first enumerates all the cases that do not satisfy the pattern and then attaches the original pattern at end of the new pattern. Revising the above example, the new pattern essentially attempts to skip an input that matches the prefix pattern ([^A]|A[^U]|AU[^T]|AUT[^H]|AUTH[^\s]|AUTH\s[^\n]{*0,99*}\n)*. When it comes to match the suffix *AUTH*\s[^\n]{*100*}, we know that it must be able to match the pattern.

There are over 20 similar expressions in the Snort rule sets. When applying this rewrite technique to them, we observe significant effects on DFA reduction. The relevant simulation results are reported in Section 6.2.

Rewrite rule (2)

As we mentioned previously, for those patterns that start with '^' and contain classes of characters with length restrictions, a large value of the length restriction can still cause the creation of complex DFA. A significant number of patterns in the SHORT 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 6, \s overlaps with ^\n. Consequently, in an input string, white space characters cause ambiguity of whether they should match \s+ or be counted as part of the 1024 non-return characters [^\n]{*1024*}. Hence, the DFA needs 1024^2 states to remember all the possible consequences that these white spaces could lead to, so that later after 1024 characters, we can check whether counting them as ^\n would match the expression; if not, then we take them as \s.

As we investigate possible solutions, we notice that NFA-based approaches such as those used in L7-filter and Snort systems are inappropriate here as this pattern can result in the creation of large numbers of active states. Specifically, the corresponding NFA contains 1042 states; among these states, one is for the matching of *SEARCH*, one for the matching of \s+, and the rest 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 one by one for the next active states.

This problem can be fixed using our second rewrite rule. Again we illustrate this rule using the pattern "^*SEARCH*\s+[^\n]{*1024*}". Under the rewriting definition, this pattern is converted to the following two patterns that together form an equivalent to the original pattern:

"^*SEARCH*\s[^\n]{*1024*}" and

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

The new pattern "^*SEARCH*\s[^\n]{*1024*}" specifies that after matching a white space, we start counting for [^\n]{*1024*} no matter what the content is. In contrast, the new pattern "^*SEARCH*\s+[^\s][^\n]{*1023*}" requires that after matching SEARCH, we skip all white spaces until we match a non-white space character [^\s]. If we do not see a return character in the next 1023 characters, it's a buffer overflow attempt. In either of the two patterns, ambiguity does not exist. Given a series of white spaces, they each have a single active state and thus the processing speed would be high.

In reality, a server implementation of a search task using "^*SEARCH*\s+[^\n]{*1024*}" usually interprets the pattern in one of the two ways: either skip a white space and take the following up to 1024 characters to conduct a search, or skip all white spaces and take the rest for the search. Hence, if we know which implantation a server uses, we can simplify our rewrite to just include one of two patterns. If we are not aware of the server implementation, we then include both patterns in the rewrite result to remain faithful to the original pattern.

Simulation results of applying this rewrite rule to the Snort rule set are reported in Section 6.2.

5 Size of DFA for Multiple Patterns and Grouping Algorithm

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 can fit in memory. As we mentioned in Section 3, 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. Creating such composite DFA is key to improving the regular expression matching speed, once the DFA for individual patterns are guaranteed to fit in memory.

In this section, we study the interaction of multiple patterns when they are compiled into a composite DFA. In some cases, the composite DFA is equal to or smaller than sum of the individual DFA. Thus, it can improve the processing speed without causing extra memory usage. In some other cases, however, the composite DFA may experience exponential growth in size, although none of the individual DFA have an

exponential component. In the following, we present two examples illustrating these cases, and 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.

5.1 Interactions of Regular Expressions

When patterns share prefixes, some states can be merged. For example, states 1 and 2 shown in Figure 9 are shared by *ABCD* and *ABAB*. Combining these patterns can save both storage and computation.

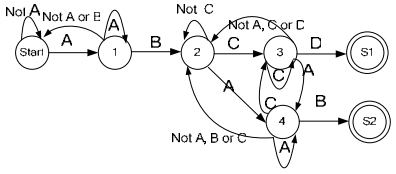

Figure 9. A DFA for pattern *ABCD* and *ABAB*

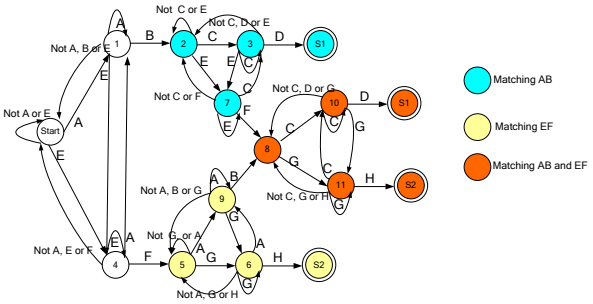

Figure 10. A DFA for pattern *AB.*CD* and *EF.*GH*

However, if patterns do not share the same prefix, putting m patterns together may generate 2^m states. Figure 10 shows a composite DFA for matching *AB.*CD* and *EF.*GH*. This DFA contains many states that did not exist in the individual DFA. 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.

5.2 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 DFA, each DFA has tens to hundreds of states. The total number of states is 9795. When we start to group multiple patterns into a composite DFA (we select patterns with a random order), the processing complexity decreases, as shown by the decreasing solid line in Figure 11. 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 11. 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 11) 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.

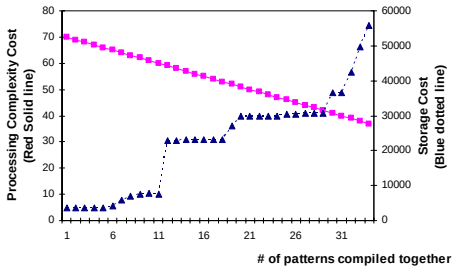

Figure 11. DFA Size and Processing Complexity of Multiple Patterns (Unsorted order)

5.3 Regular Expressions Grouping Algorithms

As shown in the previous section, some 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 12. Then we run the NFA to DFA conversion algorithm and the DFA minimization algorithm to obtain the composite DFA.

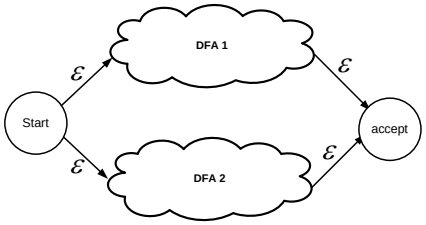

Figure 12. Composite NFA for two DFAs

We use the information on pairwire 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, it is likely that the composite DFA of R_1, R_2, R_3 will not exceed the sum of individual ones. The counter example only occurs when the regular expressions share the same prefix. In such cases, the savings from sharing the prefix may hide the effect of exponential interaction to some extent. When more patterns with the same prefix are grouped together, the exponential component will grow larger and its effect can outweigh the savings from prefix sharing. Such cases, however, are quite rare in networking applications where prefix sharing is uncommon, because each regular expression is mostly written for one specific protocol or one attack. We will validate this point using empirical results in Section 6.3.1.

We devise two grouping algorithms to be used for different processor architectures: one is for the general processor architecture where the memory is shared among all DFA, and the other is for multi-core processor with separate local memory for each core.

Multi-core architecture. We first present the algorithm for the multi-core architecture since it is simpler. In this architecture, there are multiple processing units which can run in parallel. The number of processing units 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 DFA. 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 [20]. 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 do not interact with it into the same group. We keep adding patterns until the composite DFA would be 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 which 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 DFA to be run, the processor executes each of them sequentially. Usually all the DFA are kept in the main memory for the performance purpose. Since the memory is shared among all DFA, 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.

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).

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 Multi-Processor Architecture

We evaluate the effectiveness of the proposed algorithms and report on the results in Section 6.3.2.

6 Evaluation Results

We implemented a DFA scanner with rewriting and grouping functionality for efficient regular expression matching. Based on this implementation, 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 an ungrouped DFA-based scanner and a best-known NFA-based scanner. Compared to the ungrouped DFA-based approach, our grouping approach has 362%-536% performance improvements. Compared to a state-of-the-art NFA-based implementation, our DFA-based scanner is 28-1192 times faster on traffic dumps obtained form MIT and Berkeley networks.

6.1 Experimental Setup

To focus on the regular expressions commonly used in networking application, we selected the following two 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.

We used two sets of real-world packet traces. The first set is the intrusion detection evaluation data set from the MIT DARPA project [21], which 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.

We modified Flex [22] to convert regular expressions into DFA. In our implementation, we optimized the matching algorithm by eliminating backtracking operations [22].

All the experimental results reported in the following sections were obtained on PCs with 3Ghz CPU and 4GB memory.

6.2 Effect of Rule Rewriting

We applied our rewriting scheme presented in Section 4.2 to the Linux L7-filter and SNORT pattern sets. For the Linux L7-filter pattern set, we did not identify any pattern that needs to be rewritten. For the SNORT pattern set, however, there are 38 rules that need to be rewritten. For these patterns, we gained significant storage savings as shown in Table 4. For both types of rewrite, the DFA size reduction rate is over 98%.

Table 4. Rewriting effect on the SNORT ruleset

Type of Rewrite	Number of Patterns	Average length restriction	DFA Reduction Rate
Rewrite Rule 1:	19	344	>99%[1]
Rewrite Rule 2:	17	370	>98%

19 regular expressions contain a fixed string followed by a length restricted class of characters. Since they do not start with '^', they can be rewritten using rewrite rule 1. Before rewriting, most of them generate large DFA that cannot even be compiled successfully. As stated in Section 4.1, this is because the length restriction is usually so large that the DFA fails to enumerate all the possibilities for a long

[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 using those successful ones.

sequence. We tested on a typical pattern *AUTH*\s[^\n]{*k*} with *k* varying from 2 to 1024. As Figure 13 shows, the number of states in the DFA before rewriting grows exponentially, while the number of states after rewriting grows only linearly.

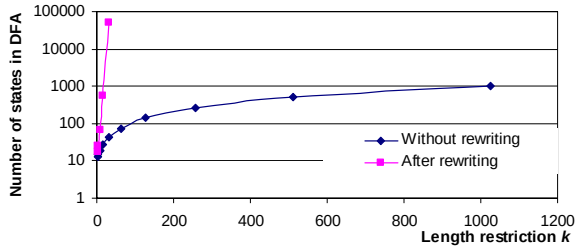

Figure 13. The number of DFA states generated for the pattern *AUTH*\s[^\n]{*k*}

The other 17 patterns belong to the category for which rewrite rule 2 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 we discussed in Section 4.2, this type of pattern generates DFA that expand quadratically with respect to the length restriction. Figure 14 validates the quadratic effect using the results from varying the length *k* for "^*SEARCH*\s+[^\n]{*k*}". Note that, when *k* is 1024 or higher, the DFA creates too many states to fit into the memory. Therefore, the figure does not show any results when $k \geq 1024$.

After applying our rewrite rule, the number of states generated is linear in *k*, as shown in Figure 14. As we explained before, in order to preserve the semantics of the pattern, "^*SEARCH*\s+[^\n]{*1024*}" is written into two patterns "^*SEARCH*\s[^\n]{*1024*}" and "^*SEARCH* \s+[^\s][^\n]{*1023*}", although in reality only one pattern is needed for one type of server implementation. Even if we use both patterns, as Figure 14 shows, the sum of total states after rewriting grows slowly compared with the number of states before rewriting.

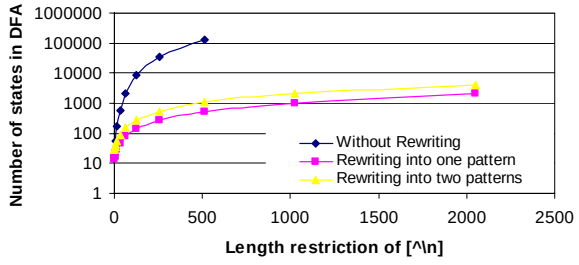

Figure 14. The number of DFA states generated by ^*SEARCH*\s+[^\n]{*k*}

With our rewriting techniques, the collection of DFA created for all the patterns in the SNORT system can fit into 95MB memory, which can be satisfied in most PC-based systems.

6.3 Effect of Grouping Multiple Patterns

In this section, we apply the grouping techniques to regular expressions sets. We show that our grouping technique can significantly boost the system throughput. Note that we do not apply grouping techniques to the SNORT rule set because most rules are only related to packets with specific header information. The patterns for the Linux L7-filter, however, can be grouped together, because the payload of an incoming packet needs to be compared against all the patterns to detect whether it contains any strings representing layer 7 protocols, regardless of the packet header information.

6.3.1 Interaction of Patterns

A majority of patterns are non-interactive in the Linux L7-filter set. As a result, most patterns in this ruleset 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 5.3, 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 sizes of individual ones. We verified this assumption with the patterns from the L7-filter set. Table 5 shows that this assumption is true for over 99.8% of the cases.

Table 5. Interaction of regular expressions

	No-interaction Pair-wise	**No-interaction among three patterns**
Linux L7-filter	71.18%	99.87%

6.3.2 Grouping Results

When applying our grouping algorithms to the Linux L7-filter pattern set, 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 do 3 state transitions as opposed to 70 state transitions that were necessary with the original ungrouped case. This results in a significant performance enhancement, as we will demonstrate later in Section 6.3.3.

We first test our grouping algorithm for multi-core architecture where local memory is limited. As shown in Table 6, we start by limiting the number states in each composite DFA to be 617, which is the size of

largest DFA of a single pattern in the Linux L7-filter set. The actual memory cost is 617 times 256 next state pointers times *log(617)* bits each pointer, which goes to 192 KB. Considering most of the modem processors have large data cache (>0.5MB), this memory cost for a single composite DFA is very small. The algorithm generates at most 10 groups. As nowadays multi-core network processor has 8-16 engines, it is feasible to allocate each composite DFA to one processor.

Table 6. Results of grouping algorithms for multi-core architecture

Composite DFA state limit	Groups	Total Number of States	Compilation Time (s)
617	10	4267	3.265
700	8	4589	4.690
1000	8	5537	5.968
2000	5	6181	12.561
4000	4	9307	29.113
8000	4	11214	56.746
16000	3	29986	54.479

Table 7. Results of grouping algorithms for general processor architecture

Total DFA state limit	Groups	Total Number of States	Compilation Time (s)
3533	12	3371	5.602
4000	10	3753	7.335
6000	8	5528	13.189
8000	6	6972	37.098
10000	5	7280	37.928
16000	4	10966	41.870
32000	3	25215	49.976

Table 7 demonstrates the grouping algorithm for general processor architecture effectively controls the aggregated DFA states until the available memory limit. In addition, the total number of DFA states is close to the memory limit, showing the algorithm fully utilizes the memory allocated to the packet scanner. Note that we start memory limit to be 3533 DFA states which is the sum states of individual DFA. Simulation Results shows that we can group 70 patterns into 12 groups with no extra memory usage.

Table 6 and Table 7 demonstrate that both our grouping algorithms are efficient. Their running time are less than one minute. This overhead is a one time cost. In networking environment, the packet content scanner operates at 24/7 until there are new patterns inserted. As patterns in the Linux L7-filter or Snort system do not change frequently, one minute overhead is negligible.

6.3.3 Speed Comparison

We measured the throughput of our DFA-based scanner on both the MIT dump and the Berkeley dump. The results are shown in Figure 15. When patterns in L7-filters were grouped into 10 groups, although the number of memory accesses per character was decreased from 70 to 10, we did not observe a 7x performance improvement because there were other system costs involved (e.g., I/O cost). However, the DFA scanner still gained 310% – 410% performance improvements over the ungrouped DFA case. When we decreased the number of groups to 3, it gained 362%-536% performance improvements.

The performance benefits obtained from the MIT dump and Berkeley dump are different, because the MIT dump mostly contains intrusion packets that are usually 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 and do match any protocol in the L7-filter. Therefore, many DFA are inactive, so the system throughput is higher than that for the MIT dump.

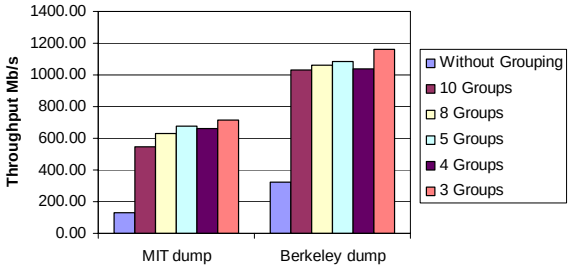

Figure 15. Throughput of the DFA scanner

Table 8. Comparison of the DFA scanner and NFA-based Systems

Throughputs (Mb/s)	MIT dump	Berkeley dump
NFA (PCRE)	0.6	40.64
DFA (70 rules)	133.36	324.48
DFA group (3 groups)	715.36	1159.2

We also compared our DFA-based algorithms with the start-of-the-art NFA-based regular expression matching implementation (PCRE[23]). Both L7-filter and SNORT systems use this NFA-based library. The results are summarized in Table 8. The NFA-based approach has poor performance on the MIT dump, because the packets are long and often large numbers of states are active simultaneously. The DFA-based algorithms consistently provide a performance improvement over the NFA-based algorithm. Even

without grouping, the DFA-based algorithm is 8-222 times faster than the NFA-based approach. When using the grouping algorithm, the DFA scanner is 28-1192 times faster than the NFA-based approach.

7 Conclusion

We consider the implementation of fast regular expression scanning for networking applications.. While NFA-based approaches are usually adopted for implementation because naïve DFA implementations can have exponentially growing memory costs, we show that with our rewriting techniques, very high performance DFA-based approaches are possible. Our DFA-based implementation yields 8-222 times performance improvement over a widely used NFA implementation. In addition, we presented a scheme that selectively groups patterns together to further speedup the matching process by a 3-5 factor. This grouping scheme can be applied to general processor architectures, where the DFA for one group corresponds to one process or thread. In addition, it can also be used for multi-core architecture or NPU architecture where groups of patterns can be processed in parallel among different processing units.

8 References

[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. 32nd Annual International Symposium on Computer Architecture (LISA)*, Madison, Wisconsin, 2005.

[5] Y. Cho and W. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," *Proc. Symposium on Field-Programmable Custom Computing Machines*, 2004.

[6] Z. K. Baker and V. K. Prasanna, "Time and area efficient pattern matching on FPGAs," *Proc. International Symposium on Field Programmable Gate Arrays (FPGA)*, 2004.

[7] Z. K. Baker and V. K. Prasanna, "A methodology for synthesis of efficient intrusion detection systems on FPGAs.," *Proc. Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[8] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speedup up intrusion detection," *Proc. Workshop on Architectural Support for Security and Anti-virus (WASSA) Held in Cooperation with ASPLOS XI*, 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*, Berlin, Germany, October, 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, 28(4)*, 2003.

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

[15] J. Moscola, J. Lockwood, R. P. Loui, and Michael Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2003.

[16] R. Sommer and V. Paxson, "Enhancing Byte-Level Network Intrusion Detection Signatures with Context," *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2003.

[17] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2001.

[18] R. Franklin, D. Carver, and B. Hutchings, "Assisting network intrusion detection with reconfigurable hardware," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2002.

[19] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[20] 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.

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

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

[23] PCRE, Perl Compatible Regular Expressions, http://www.pcre.org