A Workload for Evaluating Deep Packet Inspection Architectures

Michela Becchi, Mark Franklin, *Fellow, IEEE*, and Patrick Crowley

***Abstract*—High-speed content inspection of network traffic is an important new application area for programmable networking systems, and has recently led to several proposals for high-performance regular expression matching. At the same time, the number and complexity of the patterns present in well-known network intrusion detection systems has been rapidly increasing. This increase is important since both the practicality and the performance of specific pattern matching designs are strictly dependent upon characteristics of the underlying regular expression set. However, a commonly agreed upon workload for the evaluation of deep packet inspection architectures is still missing, leading to frequent unfair comparisons, and to designs lacking in generality or scalability.**

In this paper, we propose a workload for the evaluation of regular expression matching architectures. The workload includes a regular expression model and a traffic generator, with the former characterizing different levels of expressiveness within rule-sets and the latter characterizing varying degrees of malicious network activity. The proposed workload is used here to evaluate designs (e.g., different memory layouts and hardware organizations) where the matching algorithm is based on compressed deterministic and non deterministic finite automata (DFAs and NFAs).

I. INTRODUCTION

Network packets are increasingly classified not only by the fields of their headers, but also by the content of their payloads. This trend has been driven by a desire to detect and remove malicious messages such as viruses from interfering with network and system operation. Software tools that provide signature-based deep packet inspection include Snort [4][5], Bro [6] and ClamAV [7] and hardware-based devices for these inspections include the Cisco family of Security Appliances [8] and the Citrix Application Firewall[9].

The need for line-rate inspection of network traffic has led to several recent proposals for high performance regular expression matching [12]-[16][18]-[20][22]-[25]. While regular expressions have been extensively studied and there exists a well-known theory concerning complexity and efficiency [1]-[3], deep packet inspection imposes two requirements on regular expression evaluation which distinguish this domain from the well-established theory.

First, the operation has to be performed at line rate, currently in the range of several gigabits per second (Gbps). Second, huge pattern-sets consisting of hundreds or thousands of regular expressions must be matched. The design of a pattern matching architecture must focus on two issues: processing time and memory requirements (i.e., the space needed to store the finite automata). These issues are often closely coupled: compact data structures can be accommodated on smaller and therefore faster memories, and are likely to exhibit better cache behavior.

In the past several years, publicly available pattern-sets have increased both in size and complexity. Exact-match strings have been progressively replaced by patterns containing character ranges, wildcards, and Perl-compatible regular expressions. These changes have two consequences. First, extremely large data structures, sometimes in the range of gigabytes, are required to accommodate current rule-sets. Second, many solutions described in previous work only address restricted regular expression classes (e.g.: exact-match strings) and are not suitable for more complex and general patterns. Since proposals in the research literature have appeared in the midst of changing rule-set characteristics, it is often difficult to judge whether a previously proposed solution still has relevance for contemporary workloads.

To see why this is important, we note that DFAs have been used extensively to perform regular expression matching, particularly in recent research papers, because they require processing time linear in the number of input characters, independent of the number of patterns to be matched. In particular, compression techniques to reduce the memory space

Michela Becchi is with the Computer Science and Engineering Department, Washington University in St. Louis, St. Louis, MO 63130 USA (phone: 314-935-4306; fax: 314-935-7302; email: mbecchi@cse.wustl.edu).

Mark Franklin is with the Computer Science and Engineering Department, Washington University in St. Louis, St. Louis, MO 63130 USA (email: jbf@cse.wustl.edu).

Patrick Crowley is with the Computer Science and Engineering Department, Washington University in St. Louis, St. Louis, MO 63130 USA (email: pcrowley@cse.wustl.edu).

978-1-4244-2778-9/08/$25.00 ©2008 IEEE

required to store DFAs have been recently proposed [15][19]. However, a single DFA is impractical if the pattern-set contains simple wildcard repetitions [20] since an excessive number of states is now required to encode the DFA. Therefore, solutions using a single DFA are only applicable either on simple regular expressions not containing wildcard repetitions, or on small pattern sets containing at most a dozen generic regular expressions. Unfortunately, this consideration is often omitted in the literature.

A critical element missing from the computer and networking systems community is an accepted, standard workload that allows for a fair comparison between different regular expression data structures and architectures. The workload should consist of two parts: a regular expression model and a traffic model. The former is needed to evaluate the size and the complexity of the required data structures, and the latter to assess the performance of the design under different types of network traffic (i.e., different degrees of malicious activity). Additionally, since new virus detection rules are continuously created, not only should the regular expression model reflect the characteristics of current pattern-sets, but it should also allow projections into the future.

In this work, we start by articulating the characteristics of current real-world rule-sets. These observations are then used to create a regular expression model that can be used to generate synthetic rule-sets with varying characteristics (Section II). In Section III, we describe in greater detail the use of NFAs and DFAs to perform regular expression matching. Note that NFAs have the important property that the number of states in an NFA-based representation of a regular expression does not exceed that of the number of characters in the pattern-set; DFAs, on the other hand, can have a number of states that is exponential in the number of pattern-set characters. In Section IV we describe the traffic model created for our workload. Since the performance of any design depends on the concrete memory representation of the finite automaton, three different memory layouts are discussed in Section V. In Section VI, the regular expression engine is evaluated on a single processor using different cache configurations. Section VII briefly considers related work and Section VIII presents our summary and conclusion.

All of the rule-sets and software described in this paper are available as open-source at our website *http://regex.wustl.edu.*

II. THE REGULAR EXPRESSION MODEL

In this section, we first categorize the features used in regular expressions found in publicly-available network intrusion detection systems and anti-virus rule-sets. Using these features, a synthetic regular expression set is developed that can be used to generate synthetic patterns reflecting characteristics of real world data.

A. Regular expression taxonomy

We begin by characterizing features found in regular expressions derived from commonly used anti-viruses and network intrusion detection systems.

Exact-match strings. Exact-match strings represent the simplest patterns which may appear in a rule-set. An exact-match string is a *fixed size* pattern which must occur in the input text exactly *as is.* Rule-sets consisting only of exact-match strings exhibit two important properties. First, DFA based solutions [3] can be effectively used since their size is dependent only upon the number of characters in the pattern set (and do not exceed that of the corresponding NFA). Second, optimizations based on hashing schemes can be exploited [16][17].

Character sets and simple wildcards. Character sets are found in regular expressions in two forms: either as $[c_1\text{-}c_jc_kc_l]$ expressions, or as special cumulative symbols, namely *\s, \d, \w, \S, \D, \W*. In the first case the set includes all characters between c_1 and c_j, c_k and c_1. In the second case the set consists of all space characters (*\s*), all digits (*\d*), all alphanumerical characters (*\w*), and their complements (*\S, \D, \W*). A *wildcard* (e.g., any character of the alphabet) is represented through a non-escaped *dot*.

Clearly, these sub-patterns allow more expressive regular expressions, each of them representing a set of exact-match strings. In general, character sets and wildcards don't allow the direct application of the Aho-Corasick DFA construction algorithm [3] and of hashing schemes [16][17]. However, at the cost of increasing the size of the pattern-set, it is possible to perform an exhaustive enumeration of the corresponding exact-match strings and produce the simpler case that doesn't violate the two "Exact-Match" properties cited above.

Simple character repetitions. Simple character repetitions appear in the form *c+* and *c*, c* being any character of the alphabet. Simple character repetitions still allow the size of the DFA not to exceed the number of characters in the pattern-set. However, in this case, using exhaustive enumeration to reduce a regular expression to a set of exact match strings is not possible since an infinite number of such strings exist. Therefore, hashing schemes such as [16] and [17] are not applicable.

Character sets and wildcard repetitions. Repetitions of character sets and wildcards (also called dot-star terms) introduce additional complexity. When several such regular expressions are compiled together into a single DFA, the DFA size may explode [20]. Thus, not only are hashing schemes not applicable to this case, but a single DFA may not be a feasible solution. A common technique is to cluster rules into multiple, concurrent DFAs [14][25]. This reduces the memory footprint in exchange for increased memory bandwidth; specifically, *N* DFAs require an *N*-fold increase in memory bandwidth. As an alternative, NFAs can be used, again trading off memory space with memory bandwidth requirements.

Counting constraints. Counting constraints appear as bounded repetitions of simple characters, sub-patterns,

character sets and wildcards. Their upper bound may or may not be constrained. In the case of simple characters and sub-patterns having repetitions with a constrained upper bound, an exhaustive enumeration of the corresponding exact-match strings is possible with the implications seen above. Particularly problematic, however, are counting constraints on large character sets and wildcards. As highlighted in [14] [20], they can lead to exponential state blow-up when performing NFA-DFA transformation even on *single* regular expressions compiled in isolation. With counting constraints, DFA-based methods are infeasible. It is preferable to replace counting constraints with unbounded repetitions and handle the value of the counter separately [20].

B. Analysis of practical rule-sets

This section presents an analysis of real-world rule-sets. Our objectives are three-fold. First, we verify that the feature set above is comprehensive. Second, we evaluate the complexity of practical rule-sets. Third, we derive a methodology for building synthetic rule-sets. Synthetic rules are useful for exploring the performance of competing approaches; they can be constructed to both reflect real-world characteristics by anonymizing existing rules and to explore characteristics not yet found in existing rule-sets.

The regular expressions we study are extracted from Snort [4][5] and Bro [6], two open-source network intrusion detection systems, and from ClamAV [7], an anti-virus system intended for email scanning on mail getaways. For ClamAV, all the regular expressions from *main.db* database (August 2007) are considered. For Bro, the *ex_web* regular expressions (version 0.8) and the *sig-addendum* patterns (version 0.9) are considered. For Snort, all the rules available (versions 2.7, November 2007) are included.

It should first be noted that 5,549 out of 8,536 Snort active rules contain at least one Perl Compatible Regular Expression (PCRE). However, half of those regular expressions contain just counting constraints on wildcards that are better handled using counters rather than through finite automata. Therefore, those expressions were excluded from our analysis.

TABLE 1: STATISTICS ON THE LENGTH OF REGULAR EXPRESSIONS AND THEIR EXACT-MATCH SUB-PATTERNS FOR SEVERAL RULE-SETS.

Data-set	# Reg -Ex	Length			Sub-pattern length		
		min	max	avg	min	max	avg
Snort1	22	4	102	25.3	1	22	5
Snort2	78	11	101	33.6	1	28	6.2
Snort3	102	1	52	21.2	1	22	5
Snort4	468	4	393	27.9	1	46	10.1
Bro0.8	226	1	84	12.3	1	46	4.1
Bro0.9	40	4	45	19.4	1	43	11.3
ClamAV	30411	6	210	67.2	1	210	64.3

Furthermore, we note that within Snort, packet payload inspection (requiring regular expression matching) is performed only after packet header filtering. A useful partitioning of the remaining set of Snort rules can therefore be based on clusters of rules sharing the same header information. As a result, Snort regular expressions are associated with one of the following four header sets: 1) (tcp, $external_net, any, $home_net, $http_ports), 2) (tcp, $home_net, any, $external_net, 25), 3) (tcp, $home_net, any, $external_net, any), and 4) (tcp, $home_net, any, $external_net, $http_ports).

The set of regular expressions can now be decomposed into a number of exact-match sub-patterns separated by character sets (single and repeated), wildcards (single and repeated), counting constraints, and disjunctions of simple sub-patterns. With this limited feature set, every regular expression can be examined and aggregate feature statistics can be collected including: minimum, maximum and average pattern length, minimum, maximum and average exact-match sub-pattern length, number of character sets and of their repetitions, number of wildcards and dot-star terms, number of sub-pattern repetitions, number of counting constraints (classified by type), and number of disjunctions. The results are shown in Table 1 and Table 2.

As can be observed, Snort rule-sets are the most complex in that they contain all the above sub-pattern types. Moreover, note the high occurrence of wildcard and character range

TABLE 2: CHARACTERISTICS OF PRACTICAL DATA-SETS FROM SNORT, BRO AND CLAMAV.

Data-set	# Reg -Ex	\x	\x+	.	.*	$[c_1..c_n]$	$[c_1..c_n]$*	[^\n\r]*	(sp)+	c+	c{n}	$[c_1..c_n]$ {n}	.{n}	(sp) {n}	OR-exp
Snort1	22	4\s	13\s+	4	2	3	10	8	4	0	0	0	1	5	46
Snort2	78	1\d	173\s+ 28\d+	1	18	1	1	81	0	0	2	1	0	0	0
Snort3	102	9\s 2\d	165\s+ 99\d+ 1\w+	2	5	5	3	26	1	2	1	1	0	2	18
Snort4	468	4\s	77\s+ 26\d+ 1\S+	14	38	5	9	468	7	3	0	11	3	7	87
Bro0.8	226	0	0	0	10	1399	0	0	0	0	0	0	0	8	0
Bro0.9	40	0	0	20	0	22	1	0	6	0	0	0	0	10	8
ClamAV	30411	0	0	0	1221	0	0	0	0	0	0	0	113	0	5

repetitions, which, as mentioned, cause state blow-up in the corresponding DFAs. In particular, a dot-star expression indicates that the separated sub-patterns must both appear in the input text in the indicated order, whereas the frequently occurring *[^\n\r]** term adds the additional constraint of belonging to the same line. Bro rules appear to be simpler, even though they contain some dot-star terms and character ranges. Finally, ClamAV regular expressions consist of relative long exact-match sub-patterns occasionally separated by dot-star conditions and counting constraints on wildcards.

Nearly all the regular expressions presented in the analyzed rule-sets are covered by the above description. A small exception consists of 70 Snort PCREs containing back references, a specific Perl feature that is not part of the regular expression definition. Given the above characterization of regular expressions, we now derive a set of synthetic regular expressions that may be used in performance analysis studies.

C. Generation of synthetic rule-sets

The above characterization was used in order to create a model for generation of synthetic rule-sets. The parameters of the model are the following:

- *RE#*: Number of regular expressions to be generated.
- *min_length, max_length, avg_length*: Minimum, maximum and average regular expression length.
- $f_{\backslash s}$, $f_{\backslash d}$, $f_{\backslash w}$, $f_{\backslash S}$, $f_{\backslash D}$, $f_{\backslash W}$: Frequency of special character sets.
- $f_{\backslash s+}$, $f_{\backslash d+}$, $f_{\backslash w+}$, $f_{\backslash S+}$, $f_{\backslash D+}$, $f_{\backslash W+}$: Frequency of special character set repetitions.
- $f_{[c1..ck]}$, $f_{[c1..ck]+}$, $f_{[^\backslash n\backslash r]*}$: Frequency of character set and character set repetitions ($[^\backslash n\backslash r]^*$ terms are treated separately).
- $f_{.}$, $f_{.*}$: Frequency of wildcard and dot-star terms.
- f_{c+}, $f_{(sp)+}$: Frequency of simple character and sub-pattern repetitions.
- $f_{c\{n\}}$, $f_{(sp)\{n\}}$, $f_{[c1..ck]\{n\}}$, $f_{.\{n\}}$: Frequency of counting constraints on simple characters, sub-patterns, character ranges and wildcards.
- f_{OR}: Frequency of disjunction sub-patterns.
- *min_sp_length, max_sp_length, avg_sp_length*: Minimum, maximum and average exact-match sub-pattern length.
- S_{EM}: Set of available exact-match sub-patterns.

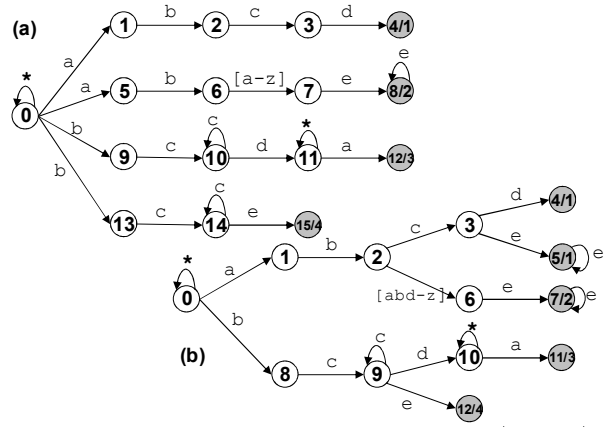

Figure 1: NFA accepting RegEx: (1) *abcd*, (2) *ab[a-z]e+*, (3) *bc+d.*a*, (4) *bc+e*. Accepting states are represented in gray (the number after the slash indicates the accepted regular expression). The original NFA is represented on the left, whereas its compressed form obtained by collapsing common prefixes is shown on the right. State 0 has an auto-loop since the match is not anchored to the beginning of the input string, but may occur at any position.

The last two items are mutually exclusive. Regarding exact-match sub-patterns, two different approaches are possible. In the first, sub-patterns are automatically generated by concatenating randomly selected characters with the length of each sub-pattern being a random variable. This random variable is either uniformly distributed between *min_sp_length* and *max_sp_length*, or normally distributed with mean *avg_sp_length* and variance dependent upon *min_sp_length* and *max_sp_length*. A similar statistical approach can also be applied to the character selection process thus allowing biasing of the alphabet set used. The drawback of this method is that it provides no direct control on whether the generated strings have real meaning. A second approach, the one chosen for our synthetic rule-sets, utilizes sub-patterns derived from real world data-sets (e.g., segments of network protocols or URLs) and collected into S_{EM}.

Given the selected parameters, the regular expression generator operates as follows. From *RE#* and the above frequencies, the generator computes the set of available non-exact match terms S_{NEM} and the average number of non-exact match terms per regular expression N_{NEM}. The length of each regular expression is randomly selected according to the length parameters listed above. Each regular expression is built by alternating an exact- and a non-exact-match sub-pattern. Those patterns are randomly selected from S_{EM} and S_{NEM} according to a uniform distribution. After selection, these two sets are updated. The concatenation process stops upon reaching the pre-computed regular expression length; several exact-match patterns may be appended after insertion of N_{NEM} non-exact match terms in the same regular expression.

III. USING FINITE AUTOMATA TO PERFORM REGULAR EXPRESSION MATCHING

High-speed regular expression matching systems are based on either NFAs or DFAs. The theory for both constructing an NFA for a given regular expression set and converting an NFA into its DFA counterpart is well known [1]. However, in some cases, NFA-DFA conversion can lead to state explosion, leaving DFA-based solutions impractical [20]. In this paper we focus on NFAs, since they allow us to explore pattern-sets with arbitrary complexity without incurring unreasonable resource requirements. However, for the sake of completeness, we briefly discuss DFAs and compare the results obtained through the use of the two automata.

A. Introduction to NFAs

In Figure 1(a) we represent an NFA accepting regular expressions *abcd*, *ab[a-z]e+*, *bc+d.*a* and *bc+e,* constructed in

the standard way [1]. To evaluate a representation, whether an NFA, DFA, or other data structure, we must consider two metrics: the amount of memory needed to store it and the amount of memory bandwidth needed to operate it.

NFA size depends only on the number of characters in the pattern-set. This is true even if some regular expressions contain simple and repeated character ranges.

To find the operating memory bandwidth, we must understand how the NFA works. The pattern matching operation starts from the entry state 0, as shown in Figure 1. A match is reported every time an accepting state (in gray) is traversed. The characters in the input text are processed in sequence, and all the outgoing transitions from the active state labeled with the current input character are taken. Notice that, since each state can have more than one transition on any given character, many states can be active in parallel. We will call these states the *active set*. Since every state traversal implies one or more memory operations, the size of the active set gives a measure of the memory bandwidth requirement and, in case of sequential memory accesses, of the processing time.

As an example, let us process the input text *abcda*. The NFA traversal will involve the following active set sequence (accepting states are underlined).

$$(0) \xrightarrow{a} (0,1,5) \xrightarrow{b} (0,9,13,2,6) \xrightarrow{c} (0,10,14,3,7)$$
$$\xrightarrow{d} (0,11,\underline{4}) \xrightarrow{a} (0,1,5,11,\underline{12})$$

In this case, the maximum active set size is 5, and the total number of state traversals is 22. The worst case traversal often reported in literature corresponds to an active set including all the states in the NFA. Notice that this worst case is in practice never achieved. As an example, state 1 can never be active together with any of 2, 3, 4, 6, 8, 9, 10, 13, 14 and 15 as it is entered upon a different input character.

In general the NFA for a given regular expression set is not unique. However, given any NFA, it is possible to compute an optimized representation of it by collapsing common paths starting at the entry state. This can be done by applying a variant of the NFA-DFA conversion algorithm detailed in [1]. Specifically, for every state, the optimization algorithm: (i) expands only the characters upon which there is at least one outgoing transition, (ii) avoids expanding self-transitions.

The compressed NFA obtained by applying the above optimization to the NFA represented in Figure 1(a) is shown in Figure 1(b). Note that the compressed variant is preferable for two reasons. First, it has a smaller size, both in terms of the number of states and transitions. Second, the active set size has lower bounds thus resulting in a lower traversal time. In fact, the input text *abcda* can be now processed through the following active set sequence.

$$(0) \xrightarrow{a} (0,1) \xrightarrow{b} (0,8,2) \xrightarrow{c} (0,9,3)$$
$$\xrightarrow{d} (0,10,\underline{4}) \xrightarrow{a} (0,1,10,\underline{11})$$

Notice that the maximum active set size has decreased to 4, and the total number of state traversals to 16. For the remainder of this paper we consider only compressed NFAs.

B. Introduction to DFAs

In a DFA, each state has *one and only one* outgoing transition for each symbol of the alphabet. As a consequence, during traversal the active set will always consist of a single state. The interested reader can refer to [1] and [2] for the details about DFA construction and operation.

As mentioned above and as discussed in [14] and [20], pattern-sets containing complex regular expressions can make a single DFA infeasible. State explosion may in fact take place when converting the corresponding NFA into its deterministic counterpart. This can be mitigated by clustering the rules into groups and compiling them into distinct DFAs [14] [25]. As will be shown in Section VI, this will increase the memory bandwidth requirement, in that all the DFAs must be accessed while processing every input character.

As will be discussed in Section VII, several techniques have been proposed to reduce the memory space requirements of a DFA. In this paper, we use the default transition based scheme proposed in [19], which, to our knowledge, represents the most effective DFA compression algorithm available. The scheme reduces the number of transitions necessary to represent a DFA at the cost of increasing the number of state traversals per character (in the worst case, by a factor 2).

C. NFAs and DFAs from synthetic regular expression sets

The regular expression model discussed in Section II was proposed to generate synthetic regular expression sets. We now study how the characteristics of the pattern-set reflect on the corresponding finite automata. In all cases, the length of the regular expressions was uniformly distributed. Two length ranges were considered, the first between 20 and 60 (selected with probability ¾) and the second between 20 and 100 (selected with probability ¼).

The generated data-sets consist of 300 regular expressions with increasing complexity (Figure 2). In the figure, complexity increases left to right with the lowest complexity regular expressions in this set containing only exact-match terms, and the highest complexity regular expression containing a high number of dot-star terms. In particular, the second and the third expression sets contain character ranges (randomly selected between the different \x and \x+ groups), with average frequency of 0.5 and 1 per regular expression. The last three expression sets contain, in addition to one character range per regular expression, wildcard repetitions (either in the .* or in the *[^\r\n]** form) with average frequency ranging from 0.3 to 0.9 per regular expression.

For the sake of completeness, we also generated the corresponding DFAs. In the case of complex patterns, we needed to perform regular expression clustering and generate multiple DFAs (1, 2, 2, 14, 24, and 32 DFAs, going from the leftmost to the rightmost pattern-set, respectively). In particular, clustering was done by recursively bisecting the pattern-set so to keep the size of each DFA below 100,000 states.

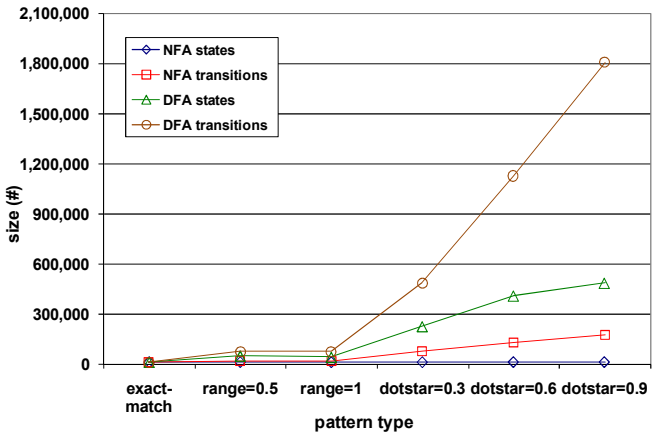

Figure 2: Number of states and transitions for pattern-sets with increasing complexity (from exact match strings, to patterns with character ranges and wildcard repetitions).

Three basic observations can be made. First, as expected, the number of NFA states is similar across the different data-sets (in that it depends only on the number and length of the regular expressions). Second, the number of NFA transitions increases with the pattern complexity. Third, the total number of DFA states, especially in case of complex patterns, exceeds by far that of NFA states and transitions. Applying the default transition construction algorithm proposed in [19] to these sets leads to 2-to-4 outgoing transitions per DFA state.

This illustrates how the workload model developed here can be tuned to effectively reflect different types of regular expression sets, and thus how one can explore new regular expression sets that may be developed in the future in response to new virus designs.

IV. THE TRAFFIC MODEL

To analyze performance, it is necessary to specify a traffic model to direct the FA (either NFA or DFA) traversal. One approach is to examine real Internet traffic and abstract from this examination a "typical" subset of malicious traffic and non-malicious traffic. This subset could then be used in conjunction with the synthetic FA and an associated architecture model to examine the effect of different architecture parameters on performance. Another approach to generating traffic is to embed in the synthetic FA a set of probabilities associated with state transitions. Traffic generation in this case involves traversing the FA by using a random number generator and associated probability distributions to determine movement from state to state. We adopt the latter approach. In particular, we have created a synthetic traffic generator that produces a plausible traffic stream given an FA and a probability p_M of experiencing malicious traffic.

To understand the operation of the traffic generator, we must first characterize good and bad traffic. Since regular expression data-sets have the goal of detecting suspicious activities, at a first glance an input stream may be considered malicious if it matches some patterns in the rule-set. In reality, however, a more dangerous attacker will harm the system by slowing down or breaking its operation without being noticed.

NFA and DFA traversals tend to exhibit a high degree of *locality* especially when guided by average traffic. In fact, average traffic tends not to match any patterns and therefore tends to limit the traversal to a restricted number of low-depth states. This locality can be exploited in order to engineer high performance packet inspection architectures, either by using caches or, in the case of ASIC designs, by accommodating the portions of the automata representing the *fast path* (i.e., the most commonly traversed states) on fast on-chip memory. Therefore, malicious traffic will force the system to operate on its *slow path* by preventing traversal locality.

Ideally, bad traffic will cause random walks in the FA. Since this goal is difficult to achieve, especially given the fact that the FA structures are not known to the attacker, a malicious strategy would be to send pieces of harmful traffic. As an example, virus signatures can be easily found on the web and in open source network intrusion and detection system (NIDS) rule-sets. Also, the input stream should not be repetitive (since this would foster traversal locality), and should avoid complete matches (which would alarm the NIDS).

The goal of the traffic generator should now be clear. The generated character stream should contain only partial matches, and cause the traversal of as many FA states as possible. One way to accomplish this is based on a simple observation made above. For any given state, *forward* outgoing transitions (that is, transitions directed toward deeper states) determine progress in the match. Therefore, malicious traffic will tend to follow forward transitions unless they lead to an accepting state. This will also limit low-depth state traversals, which dominate average traffic.

Since the FA traversal always starts at the entry state s_0 and since the FA is given to the traffic generator, the problem of incrementally generating the input stream can be formulated as follows: *given the current set of active states, what is the next character to be processed*? At each step, either a forward transition is taken with probability p_M or a random character is selected with probability 1- p_M. If the decision is to follow a forward transition, the selection of the specific transition (and, as a consequence, of the corresponding character) happens as follows. All the outgoing transitions from the active set are considered, and the one leading to the deepest state is chosen. If no forward transition is available, any character can be randomly selected.

This basic operation can be refined in several ways to simulate the behavior of more sophisticated attackers.

For example, observe that the model is memory-less and the selection of the next transition is dependent only on the current set of active states (i.e., a one step Markov chain) and is thus oblivious to the characters already sent. Despite the use of

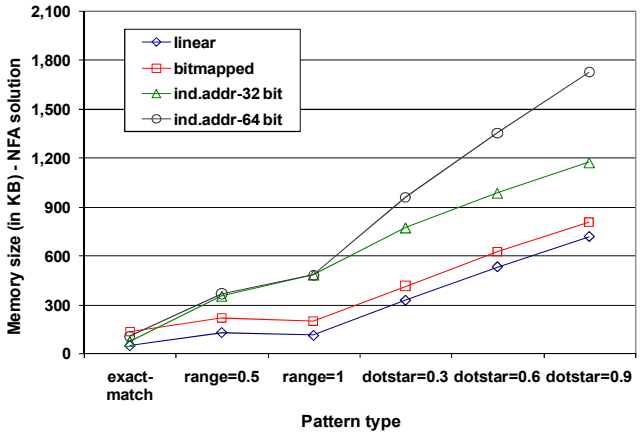

Figure 3: Memory size of the pattern-sets in Figure 2 using an NFA with different memory representations.

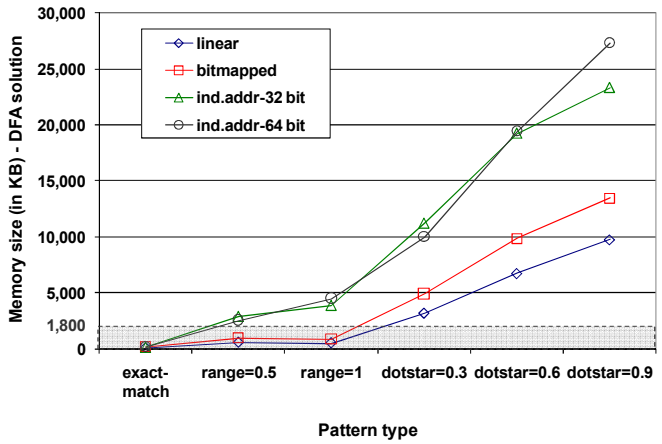

Figure 4: Memory size of the pattern-sets in Figure 2 using DFAs with different memory representations. Note the difference in scale compared to Figure 3.

randomness, this may reproduce previously generated sub-patterns. The traffic generator can be augmented with a history buffer of configurable size that records the past state traversals, making the exploration more selective.

The traffic generator is instrumented to produce streams of random length between a user-specified minimum and maximum value. In the simulations presented in Section VI, those limits are set to 5KB and 1MB, to simulate traffic ranging from small flows to file transfer operations.

V. MEMORY LAYOUT OPTIONS

An orthogonal design problem consists of selecting the concrete data structure used to perform pattern matching; such a data structure must account for every bit needed to represent the NFA/DFA.

Given an alphabet Σ of cardinality $|\Sigma|$ (256 in case of the ASCII alphabet), a naive solution consists of representing a state s as a list of $|\Sigma|$ next state pointers. However, this approach does not leverage the fact that most states have only a restricted number of outgoing transitions.

Alternatively, we consider three different techniques to encode the compressed FA: i) *linear encoding*, ii) *bitmapped encoding* and iii) *address indirection*. These encoding schemes access memory differently and thus differ in the number of memory accesses required for each state traversal. While each of these techniques can be tuned, for the sake of simplicity and comparison we assume the use of a 32-bit word aligned memory layout and avoid FA specific optimizations.

Note that an NFA state may have several outgoing transitions on the same character, which may complicate the memory layout. To avoid this, states with this characteristic are split into multiple states connected through non-consuming *epsilon* transitions. Every time a state s is activated, so are all the states connected to s via an epsilon transition. This approach, despite increasing the active set size, allows a uniform NFA state representation.

A. *Linear encoding*

If linear encoding is used, a state with l transitions is encoded through $l+1$ 32-bit words with the first one representing the epsilon/default, and the others the remaining transitions. Each word has one bit indicating whether the transition is the last one within the state, 8-bits representing the input character upon which the transition must be taken, and the remaining bits devoted to the next state address. A state traversal starts from the first word, and involves going through the transitions in sequence until the one matching the input character is found or its absence is verified. When using linear addressing, a threshold t is used. States having more that t outgoing transitions are fully represented through $|\Sigma|+1$ pointers (and accommodated in a separate memory region) to allow fast access.

B. *Bitmapped encoding*

Bitmaps [12] admit a reasonable upper bound on the number of memory accesses needed to process a character. Specifically, in this context a bitmap is an array of $|\Sigma|$ bits, each one indicating whether the corresponding transition exists or not. Each state is encoded through a bitmap and a sequence of $l+1$ memory words, each one representing a next state pointer. Upon state traversal, the bitmap is first analyzed. If it contains a 0 in the position of the input character, then only a direct access to the epsilon/default transition is performed. Otherwise, a pop-count of the number of 1s preceding the current position is made, and this information is used to directly access the proper next state transition.

Several techniques to compress bitmaps have been proposed [27]. In this paper, we use a two level organization where a first-level 32-bit bitmap is used to address a set of second-level 8-bit bitmaps. For sparse FAs, this scheme shows an acceptable overhead. Furthermore, as in the linear encoding case, states with a high number of outgoing transitions are fully encoded.

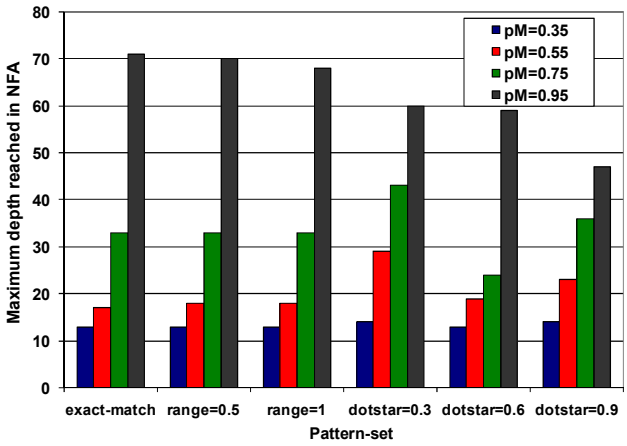

Figure 5: Effect of the traffic and the pattern-set on the maximum NFA depth reached during traversal.

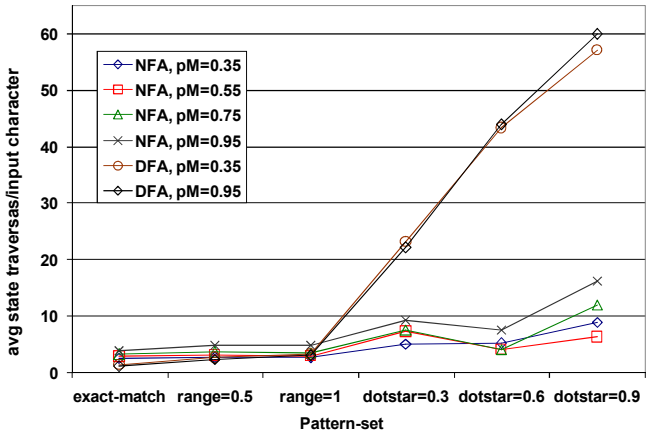

Figure 6: Effect of the traffic and the pattern-set on the average number of state traversals per input character.

C. *Indirect addressing*

Indirect addressing, a generalization of the content addressing scheme proposed in [18] for DFAs, can be used to further reduce the number of memory accesses per state traversal. With this approach each state is given an identifier consisting of: i) the list of characters upon which there exists an outgoing transition, and ii) a set of bits, called a *state discriminator*. When a state is traversed, examining the state identifier can determine whether a transition must be followed. Moreover, the order of the characters in the state identifier is used in performing a direct memory access.

The discriminator is introduced to ensure that all state identifiers are different, even for states having labeled transitions on the same set of characters. An indirection operation, performed through hashing, is thus needed in order to translate a state identifier into a memory address. Reference [18] explains how to map state identifiers to memory addresses in a manner that ensures good memory utilization.

In this paper, 8-bit discriminators are assumed. Two possible configurations are allowed: i) *32-bit* and ii) *64-bit* state identifiers. In the former case, states with more than 3 outgoing transitions must be fully represented. In the latter case, this limit is moved to 7 (as 4 more characters can be represented in the additional 32 bits).

TABLE 3: PARAMETER SPACE USED IN THE EVALUATION.

	Parameter	Values
Traffic	p_M	0.35, 0.55, 0.75, 0.95
Cache	*size*	4 KB, 16KB, 64KB, 256KB
	line	64B
	associativity	DM
	hit latency	1 clock cycle
	miss latency	30 clock cycles
Memory layout	*encoding*	linear, bitmapped, ind. addr 32-bit, ind. addr. 64-bit

Figures 3 and 4 show the memory size obtained by applying the described memory representations to the pattern-sets from Figure 2. In the case of linear and bitmapped layouts, the threshold t was set to 50 outgoing transitions. As could be expected, the linear encoding, requiring pointers only for existing transitions, is the most compact representation. Bitmapping adds the overhead for storing the bitmaps. Indirect addressing suffers since many states exceed 3/7 outgoing transitions and must therefore be fully represented. In particular, this holds when wildcards and character ranges are numerous, whereas the memory size is small in the case of simple exact-match patterns. Finally, the memory footprints of a DFA-based solution exceed by far those of an NFA representation, especially as the complexity of the regular expressions increases.

VI. EVALUATION AND DISCUSSION

In this section, we evaluate the regular expression matching operation with different cache settings. In particular, we perform our evaluation on the synthetically generated rule-sets from Figure 2. To evaluate the traversal operation, we use the traffic model presented in Section IV.

A. *Methodology*

The parameter space of our analysis is shown in Table 4. In particular, grey cells highlight parameters for which a range of settings have been tested. To perform this analysis, we created a simulator that allows the evaluation of different cache configurations and memory layouts. Specifically, given an NFA/DFA and an encoding scheme, our tool generates the corresponding memory layout (used in the previous section to compute the memory footprint). Additionally, given an input string and a memory layout, the simulator generates the sequence of memory reads triggered during the traversal. Those memory references are used as inputs to a cache simulator in order to derive latency information.

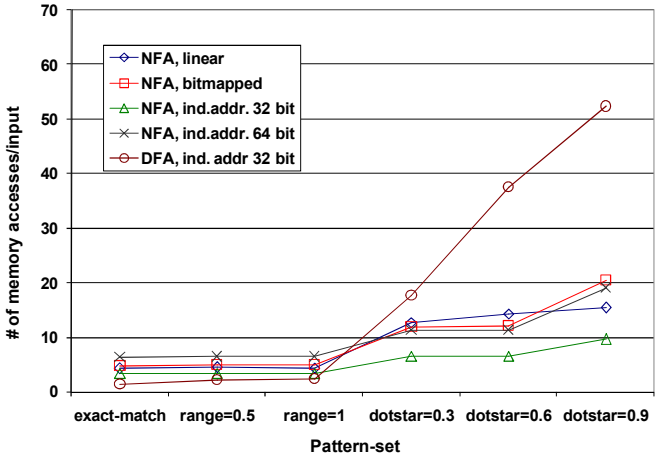

Figure 7: Effect of memory encoding on number of memory accesses per input character, p_M=0.35.

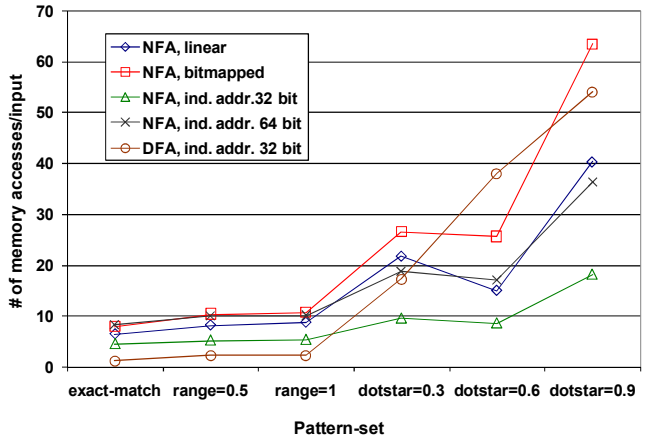

Figure 8: Effect of memory encoding on number of memory accesses per input character, p_M=0.95.

The cache size has been varied to cover values used on general purpose processors as well as on embedded multiprocessor architectures such as Tensilica Xtensa [28]. Traffic traces were generated using 10 different seeds and the results were averaged.

In general, the evaluation metrics can be grouped into the following categories:

- Traffic and finite automaton dependent: number of state traversals/input character;
- Memory encoding dependent: memory size, number of memory accesses per input character;
- Cache configuration dependent: cache hit rate, number of clock cycles per input character.

We now consider some important results of our analysis.

B. *Effect of complexity and malicious traffic*

First of all, we want to analyze the impact of the traffic pattern on the NFA/DFA traversal across the different pattern-sets. In particular, Figures 5 and 6 highlight how the maximum depth reached during the traversal and the average number of state traversals per character change when p_M increases. Recall that high values of p_M are used to model the likelihood of malicious traffic.

As could be foreseen, higher values of p_M force the traversal into deeper areas of the NFA (the same has been observed in the DFA case). More importantly, this parameter has a high impact on the percentage of states involved in the pattern matching operation: *high values of p_M imply less traversal locality*. In fact, our data show that the percentage of states traversed increases with p_M across all the pattern-sets, ranging from about 3.9% to 30% when p_M varies between 0.35 and 0.95. Moreover, NFAs outperform DFAs when the pattern complexity increases. Finally, the number of DFA state traversals is affected principally by the number of DFAs (p_M has an additional minor effect on the overhead due to following default transitions).

C. *Effect of memory encoding*

As explained above, different memory encodings imply a different number of memory accesses per state traversal. In Figures 7 and 8 we study this effect across pattern-sets. In particular, we show the results for the minimum and maximum

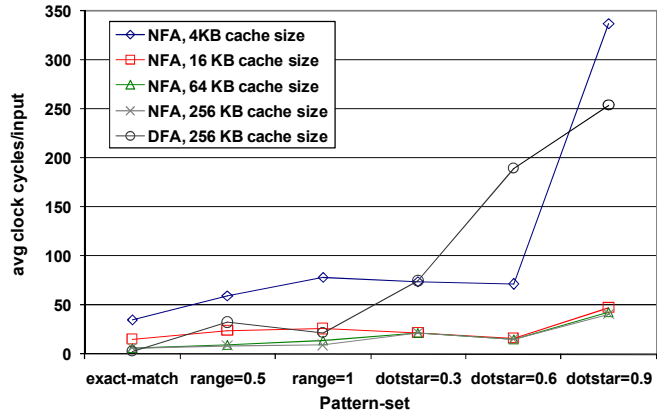

Figure 9: Effect of cache size on performance – linear encoding, p_M=0.95

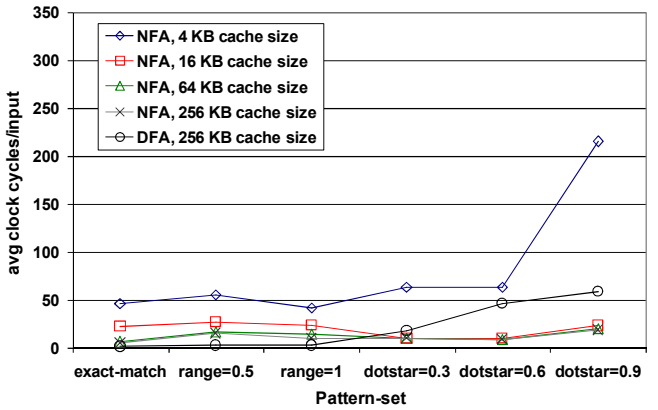

Figure 10: Effect of cache size on performance – indirect addressing (32 bit), p_M=0.95

values of p_M considered so far, namely 0.35 and 0.95.

As a general observation, the curves follow the same trends as found in Figure 6. In the case of linear addressing, the need for sequentially accessing the state transitions introduces an overhead. In the case of bitmapping, the overhead is due to the need for querying the bitmap.

As can be observed, the indirect address encoding with 32-bit identifiers has the best behavior. In fact, it requires only one memory access per state (note that in the case of epsilon transitions two accesses may be needed). Indirect addressing with 64-bit state identifiers doubles the number of needed memory accesses. In the case of bitmapping, the overhead introduced when accessing the bitmap does not justify its use compared to a simple and more compact linear encoding approach. Finally, for complex patterns even the best DFA representation (32-bit indirect addressing) leads to far worse results than the NFA counterpart, especially in the case of average traffic (low p_M).

D. Cache dependent results

Finally, we evaluate the performance of the design with different cache settings. In Figures 9 and 10 we report the average number of clock cycles per input character in the case of malicious traffic (p_M equal to 0.95). We consider two encodings: indirect addressing with 32-bit identifiers and linear encoding. In fact, the former minimizes the number of memory accesses per character and the latter has the smallest memory footprint.

Again, the trend of the curves can be compared with that in Figure 8. We can observe that, in case of NFAs, a 64KB cache is enough to ensure about one clock cycle per memory access. In fact, a 64KB cache guarantees a hit rate in excess of 98% for both encodings. However, for small cache sizes (e.g. 4KB) the performance of linear encoding approaches that of indirect addressing. In fact, the small memory footprints of the linear layout allow a higher hit rate which compensates for the worse behavior in terms of the number of memory accesses per character. The largest memory footprints of a DFA solution make a 256KB cache necessary to achieve acceptable performance (comparable to a NFA with 16KB-64KB cache).

VII. RELATED WORK

Regular expression matching at line rate has been recognized as an important problem and has been considered in related work. Prior work in this area takes two generally distinct directions: FPGA based implementations [22]-[25] and approaches suitable for deployment on a general purpose processor or on ASIC hardware [12]-[16][19][25]. Our work has the goal of proposing an evaluation methodology for the second class of solutions. We address the reader interested in benchmarking and evaluation of FPGA based designs to the work by Clark and Schimmel presented in [26].

A substantial body of research work focused on compression techniques aiming at reducing the amount of memory needed to represent DFAs. In particular, Kumar *et al.* [15] proposed an algorithm to compress a DFA through the introduction of default transitions. Their work is based on the idea of trading of memory storage requirement with processing time. A more general and less complex algorithm to achieve the same goal has been recently proposed by Becchi *et al.* [19]. By allowing only backward directed default transitions, a better memory bandwidth requirement is achieved with the same compression degree of [15].

A different category of compression techniques based on the concept of hashing and on probabilistic data structures such as Bloom Filters are proposed in [16] and [17]. However, it must be stressed that those techniques are applicable only to exact-match strings, or to regular expression classes which allow an exhaustive enumeration of the underlying patterns. As seen, none of the practical data-sets analyzed in Section II exhibits those characteristics.

Finally, a fair comparison of NFA based designs is missing. This work provides evaluations of such designs.

VIII. CONCLUSION

This paper introduces a benchmark and workload for the evaluation of regular expression architectures. We provide an objective workload and methodology for the fair evaluation of deep packet inspection (DPI) architectures. Our approach incorporates real-world regular expression rule-sets drawn from popular network security systems, along with a procedure for generating synthetic rule-sets, which can be used to explore how a given representation for regular expressions may be sensitive to changing rule characteristics. Our goal is to accelerate the design of superior regular expression data structures, algorithms, and architectures by providing an open-source evaluation framework.

The primary contribution of this paper lies in the analysis of real-world rules and in the construction of the benchmark and evaluation framework. Additionally, our analysis illustrates that NFA-based solutions are far more practical than previous work has suggested.

In this paper, we used the workload to perform a thorough evaluation of DFA- and NFA-based solutions on a processor – based architecture making use of caches. The analysis shows how different factors contribute to the performance of the regular expression matching architecture. Specifically, the behavior of the system depends on: the complexity of the underlying pattern-set, the amount of malicious activity in the traffic, the memory encoding scheme and cache size.

Finally, all of the data and software needed to recreate the results presented in this paper are available as an open-source software distribution at *http://regex.wustl.edu*.

ACKNOWLEDGEMENTS

This work has been supported by National Science Foundation grants CCF-0430012 and CCF-0427794.

REFERENCES

[1] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison Wesley, 1979.

[2] J. Hopcroft, "An nlogn algorithm for minimizing states in a finite automaton", in Theory of Machines and Computation, J. Kohavi, Ed. New York: Academic, 1971, pp. 189-196.

[3] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search", in Communications of the ACM, 1975.

[4] M. Roesch, "Snort: Lightweight Intrusion Detection for Networks", in System Administration Conf., 1999

[5] Snort: http://www.Snort.org/

[6] Bro: http://bro-ids.org/

[7] ClamAV: http://www.clamav.net/

[8] Cisco Systems. Cisco ASA 5505. http://www.cisco.com. 2007.

[9] Citrix Systems. Citrix Appl. Firewall. http://www.citrix.com. 2007.

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

[11] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context", in CCS 2003.

[12] N. Tuck *et al.*, "Deterministic memory-efficient string matching algorithms for intrusion detection", in Infocom 2004.

[13] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention", in ISCA 2005.

[14] F. Yu *et al.*, "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection", in ANCS 2006

[15] S. Kumar *et al.*, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection", in ACM SIGCOMM 2006.

[16] S. Dharmapurikar and J. Lockwood, "Fast and Scalable Pattern Matching for Content Filtering", in ANCS 2005

[17] S. Kumar *et al.*, "HEXA: Compact Data Structures for Faster Packet Processing", in ICNP 2007

[18] S. Kumar *et al.*, "Advanced Algorithms for Fast and Scalable Deep Packet Inspection", in ANCS 2006

[19] M. Becchi and P. Crowley, "An Improved Algorithm to Accelerate Regular Expression Evaluation", in ANCS 2007

[20] M. Becchi and P. Crowley, "A Hybrid Finite Automaton for Practical Deep Packet Inspection", in CoNEXT 2007

[21] R. W. Floyd, and J. D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits", in Journal of ACM, vol. 29, 1982.

[22] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs", in FCCM 2001

[23] C.R. Clark and D.Schimmel, "Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns," in FLP 2003.

[24] J. Moscola *et al*., "Implementation of a content-scanning module for an internet firewall," in FCCM 2003.

[25] B. Brodie *et al.*, "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching," in ISCA 2006.

[26] C. R. Clark *et al*., "Modeling the Data-Dependent Performance of Pattern-Matching architectures", in FPGA 2006

[27] G. Varghese, "Network Algorithms: An Interdisciplinary Approach to Designing Fast Networked Devices", Morgan Kaufmann, 2004.

[28] http://www.tensilica.com