Sampling Techniques to Accelerate Pattern Matching in Network Intrusion Detection Systems

Domenico Ficara, Gianni Antichi, Andrea Di Pietro, Stefano Giordano, Gregorio Procissi, Fabio Vitucci
Dept. of Information Engineering, University of Pisa, ITALY
Email: < first.last > @iet.unipi.it

***Abstract*—Modern network devices need to perform deep packet inspection at high speed for security and application-specific services. Instead of standard strings to represent the dataset to be matched, state-of-the-art systems adopt regular expressions, due to their high expressive power. The current trend is to use Deterministic Finite Automata (DFAs) to match regular expressions. However, while the problem of the large memory consumption of DFAs has been solved in many different ways, only a few works have focused on increasing the lookup speed. This paper introduces a novel yet simple idea to accelerate DFAs for security applications: payload sampling. Our approach allows to skip a large portion of the text, thus processing less bytes. The price to pay is a slight number of false alarms which require a confirmation stage. Therefore, we propose a double-stage matching scheme providing two new different automata. Results show a significant speed-up in regular traffic processing, thus confirming the effectiveness of the approach.**

***Index Terms*—DFA, Intrusion Prevention, Deep Packet Inspection, Regular Expressions, Packet Classification**

I. Introduction

The last years have witnessed a great interest in the area of network security, in particular about Intrusion Detection Systems, which focus on deep packet inspection techniques. Traditionally, such techniques used common multiple-string matching algorithms, but nowadays well known tools, such as Snort and Bro, and network devices, such as those of Cisco, have adopted regular expressions (regexes) to represent string sets, due their increased expressiveness [1].

Typically, to implement regex matching, finite automata (FAs) are employed. More precisely, Non-deterministic FAs (NFAs) are representations which require several state transitions per character, thus having a time complexity for lookup of $O(m)$, where m is the number of states in the NFA. However, their main advantage is the extreme space-efficiency. Instead, DFAs require one state traversal per character only, but the price to pay for the speed advantage is an excessive amount of memory. Thus, in order to make DFAs feasible for implementation in real deep packet inspection devices with a very limited amount of memory, a large number of works have addressed memory consumption ([2][3][4][5]).

Instead, less than a handful of solutions only ([6][7]) have been proposed to increase the speed of automata, which is still limited by the processing required by every single byte. The previous works proposing acceleration techniques rely on multiplying the amount of bytes (strides) processed per cycle, with the obvious problem of memory blow-up (due to the exponential growth of edge numbers with the stride size).

Our approach to the finite automata speed up is completely innovative: sampling the text, thus having less symbols to process. Clearly, sampling introduces some issues and a certain probability of false alarms is introduced. We address these issues by using together a "sampled" DFA and a "reverse" DFA, two different versions of the original automaton. We perform a first fast search on the traffic through the sampled DFA, which is able to exclude most of non–malicious traffic, and, if necessary, a more accurate processing through the reverse DFA is triggered, in order to confirm a match.

The remainder of the paper is organized as follows. In section II related works about DFAs are discussed, while section III presents the motivations and the main idea of our proposal. Then, section IV introduces basic theoretical results on DFA sampling and section V accurately explains the overall technique. Finally, section VI shows the experimental results.

II. Related works

As mentioned above, modern systems base pattern matching on regexes through DFAs and NFAs. While the former has predictable (yet large) memory consumption and a single memory reference per character, the latter consume lower memory but require several memory accesses per symbol.

The current trend in research and industry is to use DFAs to represent regexes. Therefore, many works have been focused on memory reduction of DFAs ([2][4][8]), to make their implementation in real systems feasible (it is important to remark that all these techniques can be adopted in conjunction with our scheme in order to save memory). Instead, only a few works addressed the issue of speeding up the lookup process in DFAs, which is the target of this paper.

Basically, the idea of these previous works is to multiply the amount of bytes processed per cycle, thus working with 2, 3 or 4-byte strides. However, even observing only 2 bytes per cycle would require each DFA state to include 2^{16} transitions. To solve this problem, the authors of [7] suggest a solution by observing that in actual FAs the number of different transitions (even when k bytes are processed) is more limited. In particular, they propose the use of Equivalent Character Identifiers defining the set of input words (strides of k bytes) which produce transitions to the same next state. Moreover, Run Length Encoding is used to encode the transition table. Such an approach is not general and presents some limitations, as highlighted by [6]. Indeed, it is not feasible in contexts where big DFAs (more than 100 states) and/or large

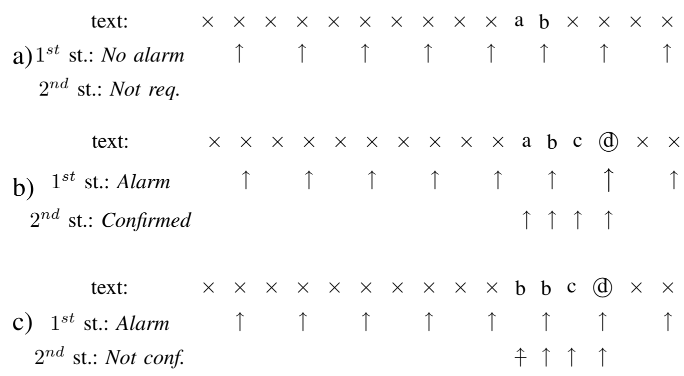

Figure 1. Example: the regex $ab.*cd$ is sampled (with $\theta = 2$) to $[ab].*[cd]$ and matched against a text of 16 bytes. Arrows point to observed chars by both stages. Sampling requires 12 memory accesses in case of a real match (b) or false alarm (c) or even 8 in the common non-matching case (a). In (c) the struck arrow points to the non-matching char which stops the second stage.

compressed alphabets are involved. Therefore, the authors of [6] try to make a k-DFA feasible by taking advantage of alphabet-reduction and default transition compression. The use of alphabet-reduction, as well as in [7], is justified by the fact that, when the number of processed bytes increases, the automaton actually uses only a small subset of the entire alphabet. Instead, the default transition compression acts by removing the transitions redundancy present in a DFA. Indeed, if the stride doubles, the number of transitions in the DFA increases quadratically, but the number of states does not; therefore, intuitively, the fraction of distinct transitions decreases and the transition redundancy tends to increase.

In this paper we propose a very innovative technique: speed up most of the processing by sampling the payload text, thus reducing the number of characters to be checked. This requires the creation of a proper automaton, the "sampled DFA", and a further processing for those strings which reveal a match. Indeed, sampling text and regexes results in a certain probability of false alarms, which, potentially, could require an additional search. The effectiveness of our solution is justified by the fact that in most cases the first (and very fast) sampled lookup is enough to classify packets, while very few packets only require a second stage of processing. While other works [9] in the area of intrusion detection already show how to sample *messages* to reduce the amount of messages to be processed in a distributed system, the application of sampling to regex-matching is a novelty and one of the main contributions of this paper.

III. Sampling DFAs

This work relies on the assumption that the majority of regular Internet traffic does not match properly written IDS regexes. Indeed, if regex sets are not poorly written (in our tests we use real and effective signatures), then a signature match will occur rarely and with malicious traffic only. Therefore we can take advantage of the fact that a match is a rare event and speed up the average case (regular traffic).

Our idea is to speed up the process by "sampling" the traffic stream: we extract a byte every θ bytes from the stream, where θ is the *sampling period*. The sampled bytes are then used as input to a proper sampled DFA. The outcome is that all regular traffic is processed θ times faster. The price to pay are false alarms: strings that would not match the original non-sampled regexes could match the sampled ones. Therefore, whenever we have a match in the sampled DFA, we have to process the suspect packet through a regular non-sampled DFA.

It is worth noticing that we aim at reducing the number of memory accesses to the memory that storing the state-machine, while we cannot reduce the number of accesses to the memory storing the packet. The reason is that even if we were interested in, say, a byte every two, accesses to packet memory do not decrease because they are performed in blocks of k-bytes long words (memory width). However, in cached-systems, this is an advantage when performing the second stage of checking: all the required characters are already available in cache, therefore the memory accesses for this stage will result in cache-hits only, thus reducing the cost of handling alarms.

A. A Motivating Example

Fig. 1 shows the principles of our scheme, with the example regex $ab.*cd$. We use a sampled DFA (that matches $[ab].*[cd]$) and a regular non-sampled one (remember that $[xy]$ means that both characters x and y trigger a transition). We perform a first check on the text by using the sampled DFA; if we find a match, we move to the second stage.

Fig. 1.a represents the common scenario with traffic that does not match signatures. It is evident, in this case, that the number of memory accesses and operations to be performed is divided by the sampling period.

Whenever the sampled regex is matched (lower two cases in figure 1, where the circled letter indicates the sample where we find the match), the non-sampled text has to be checked to confirm the match. To address this issue, the simplest and fastest way (see section V-B) is to adopt DFAs that match reversed signatures (in our example, $dc.*ba$). Any regular language is closed with respect to reversing operations [10], therefore we can always reverse a regular expression and match it inside a text by observing the text backwards from the end to the beginning. Then, if a match occurs in the second stage, because of the equivalence of reversed and forward DFAs, we have a confirmed match (fig. 1.b). Otherwise, we can claim that a false alarm occurred (fig. 1.c).

IV. Regex sampling rules

When dealing with regex matching, a simple unique condition has to be satisfied to perform a correct sampling:

Lemma 1: Let DFA A describe a single regular expression $\boldsymbol{R}$[1] and let a text T match $\boldsymbol{R}$. The corresponding sampled DFA $A^{\mathbb{S}}$ will match the sampled text $\mathbb{S}_\theta T$ if the sampling period θ satisfies the following condition:

$$\theta \le \min |s| \qquad \forall s \in \boldsymbol{R}$$

[1]Throughout the whole paper, bold letters represent regular expressions, while non-bold stand for single characters.

Proof: The proof is straightforward. In order to match the sampled text, we have to extract (by sampling) at least one char from the substring of T that matches A. Such a substring may be as short as the shortest string that belongs to the regular language described by R. Thus the condition follows. ■

Lemma 1 limits the sampling period that can be used when matching a single regex. When working with a set of regexes, the lemma still applies to the minimum length of any string matching the regexes set. Moreover the lemma states that, if the condition is satisfied, we may have false positives but we cannot have false negatives. This result is the basis of the research presented in the rest of the paper.

A. Regex rewriting

The application of sampling can be performed by rewriting regular expressions according to few simple rules. In the following we use the notation:

$$\mathbb{S}_{\theta}^{X_0}\boldsymbol{a}$$

to refer to the application of sampling to the regular language $\boldsymbol{a}$. In particular, the symbol $\mathbb{S}$ represents the *sampling* operator, θ is the sampling period, and X_0 is the position of the first sampled character. In the rest of the paper, the sampling operator $\mathbb{S}$ will be adopted also as an exponent (i.e.: $A^{\mathbb{S}}$) to denote the sampled version of a DFA.

Hereafter, we show the application of $\mathbb{S}$ to three main cases:

1) simple string str
2) concatenation of regular expressions $\boldsymbol{a}$ and $\boldsymbol{b}$: $\boldsymbol{ab}$
3) union of regular expressions $\boldsymbol{a}$ and $\boldsymbol{b}$: $\boldsymbol{a}|\boldsymbol{b}$

Sampling a string is straightforward, it consists of extracting characters at the positions defined by θ with offset X_0:

$$\mathbb{S}_{\theta}^{X_0} str = \{str(X_0 + \theta)\}$$

The offset X_0 is critical also when sampling the concatenation and union of regular expressions, it is immediate to show that:

$$\mathbb{S}_{\theta}^{X_0}\boldsymbol{ab} = (\mathbb{S}_{\theta}^{X_0}\boldsymbol{a})(\mathbb{S}_{\theta}^{X_0}\boldsymbol{b})$$

and

$$\mathbb{S}_{\theta}^{X_0}\boldsymbol{a}|\boldsymbol{b} = (\mathbb{S}_{\theta}^{X_0}\boldsymbol{a})|(\mathbb{S}_{\theta}^{X_0}\boldsymbol{b})$$

Finally, although these three cases cover all bases, it is worth discussing the case of a star closure of a character a followed by a regular expression, because of its frequent occurrence in real regex sets:

$$\mathbb{S}_{\theta}^{X_0} a * \boldsymbol{b} = a * \Big|_{i=0}^{\theta-1} \overset{i}{\mathbb{S}}_{\theta}\boldsymbol{b} = a * \mathbb{S}_{\theta}\boldsymbol{b}$$

We can easily verify the sampling of $a*$ is again $a*$. Then, since $a*$ consists of all the possible n-repetitions of a (where $n \in \mathbb{N}$), the sampling offset we apply to $\boldsymbol{b}$ can be any, hence the big **OR** operator $|$.

Finally, an example helps better understand the rules: let us sample $. * abcde * fgh$ with period $\theta = 2$. By applying the previous rules, it follows that:

$$\mathbb{S}_2[. * abcde * fgh] = . * (ac|bd)e * (fh|g)$$

This convenient and practical rule can only be applied when the length of any substring between two consecutive closures is larger than θ. However, by applying the previous rules we can always expand any regex satisfying Lemma 1 such that any substring between two closures reaches the desired length.

V. Double Stage Scheme

A. First stage: Sampled DFA

As mentioned above, the idea of DFA sampling is to speed up the string matching by extracting a byte every θ bytes from the stream and giving such characters as input to a "sampled DFA" for a first approximate search (to be subsequently confirmed). Such sampled DFA can be simply obtained by properly rewriting the regexes and building the DFA according to the new rule-set.

In details, for the process of regex rewriting, we can apply the results of section IV-A. Instead, concerning the offset X_0, which is the position of the first character to be sampled in the regex, we have to take into account all the possible starting values. This way, the resulting complete automaton can be used for string searching regardless of the point in which we start to sample the traffic.

By sampling all the regexes belonging to the set, we obtain the "sampled" rules on which the "sampled DFA" has to be built. Such a resulting automaton is a simple DFA and does not require additional information on the states or on the transitions. From this observation and as suggested by the results of section IV, we can claim that a regular language is closed with respect to the sampling operator.

However some regexes may be so short to make sampling inconvenient. For instance $\mathbb{S}_3 abc = [abc]$: although the sampled regular expression is valid, it is only 1-character long, thus potentially yielding a large number of false alarms. The good news is that these extremely short regexes are not frequent. Therefore a good and effective solution is to hardcode them, moving the matching problem from data to code and adopting a *regex_match(c)* function which is basically composed of *switch()-case* and *if-then* statements. This is a well-investigated idea [11] that is shown to be very useful with a small number of regexes. It is also compatible to our approach: the *regex_match(c)* function can still access all the bytes of the unsampled text, thus keeping the processing engine busy between two successive memory accesses to the sampled DFA. Since the number of regular expressions to be matched by such a code is small (as already pointed out, short regular expressions are fairly rare), the whole data and code required by such a function can be kept in the local cache, thus requiring no further accesses to the external memory blocks.

B. Second stage: Reverse DFA

If a matching happens in the "sampled DFA", we have to process the text again to obtain a confirmation of the match. As already mentioned, the reason is that sampling a DFA introduces a false alarm probability, since we check only a subset of the characters. If we use for this confirmation stage the original non-sampled DFA and start again from the first byte of packet, we heighten the processing and yield an excessive delay. Therefore we propose a novel scheme with a *reverse DFA*. This requires a slightly larger amount of off-line processing: all the regexes have to be independently reversed and a new DFA has to be built according to such new rules. This approach has the advantage that we can start the reverse-matching from the sample which have produced the alarm. More precisely, to take into account all the characters belonging to the string, the correct starting point for the reverse DFA is the (k+1)-th sampled char in the text: this way we process some useless characters (less than θ), but the correctness of the detection in ensured. We have devised and tested many refinements for this scheme (e.g., adopting XFA [5] and splitting the reverse DFA per regex to reduce the number of steps in the second stage), but we cannot report nor discuss them for lack of room.

Algorithm 1 Pseudo-code for the lookup procedure.

```
procedure lookup ($T$,$A^{\mathbb{S}}$,$A^R$,$\theta$)
 1: $s \leftarrow 0$
 2: while $i < length(T)$ do
 3:     $s_{next} \leftarrow A^{\mathbb{S}}[s, T[i]]$
 4:     if ($s$.acc > 0) AND ($s_{next}$.acc $\neq$ $s$.acc) then
 5:         $s' \leftarrow 0$
 6:         for $j \leftarrow i, i - \theta(s)$ do
 7:             $s' \leftarrow A^R[s', T[j]]$
 8:         while $s' == 0$ do
 9:             $s' \leftarrow A^R[s', T[j]]$
10:             $j \leftarrow j - 1$
11:         while $s' \neq 0$ do
12:             $s' \leftarrow A^R[s', T[j]]$
13:             if $s'.acc > 0$ then
14:                 Confirm Match of $s'.acc$
15:                 return to outer loop
16:             $j \leftarrow j - 1$
17:         claim False Alarm
18:     $i \leftarrow i + \theta(s_{next})$
19:     $s \leftarrow s_{next}$
```

The pseudocode for the lookup is shown in alg.1. In the listing, s represents the actual state, s_{next} is the next state and i and j are the text position we currently read respectively for the sampled DFA $A^{\mathbb{S}}$ and the reverse DFA A^R (this convenient exponent–notation will be adopted in the rest of the paper). In the pseudocode, lines 1-3 and 18-19 are part of the regular sampled DFA walk. Line 4 represents the condition we discussed above: we start a confirmation match only when we leave an accepting state (s.acc is the accepted rule) and move to a state that does not match the previous rule. This takes care of the cases where the accepting state has a loop. In lines 5-7 we initialize and start the first part of the reverse DFA walk. Then the first while loop (lines 9-10) is in charge of the cases where the first state of the reverse DFA has a closure (for instance: $(abcde*)^R = e * dcb$). Finally the next *while* loop performs the reverse DFA walk.

C. Dealing with DoS attacks

Generally, when dealing with security applications, every approach that tries to optimize a frequent case by relying on the assumptions that certain events (for us, a signature match) are relatively uncommon, is subject to be affected by aimed attacks that try to increase the probability of the rare events, thus invalidating the purpose of the method. However, as proposed in [3], such Denial of Service (DoS) attacks can be taken care of by observing the "behavior" of the incoming flows and distributing them into different queues (with different service rates) accordingly. In our scheme, the "good" or "bad" behavior of a flow is measured by the number of false alarms it generates within a time frame, since false alarms represent the largest portion of the processing cost. Therefore, according to this mechanism, flows that generate a large number of false alarms are sent to the queue with lowest service rate, while "good" flows (i.e., with few false alarms) are queued and serviced with high rates.

VI. Experimental Results

To propose verifiable and valid tests, we use the datasets of regexes of Snort intrusion detection systems and Cisco security appliances [12]. More precisely, we processed the real regex datasets with our tools for creating the new regexes (sampled and reverse), which are then parsed by the Michela Becchi's regex tool (freely available [13]) to create the corresponding DFAs (sampled and reverse).

We dumped several traffic traces of more than 10 million packets each from our department network. Such traces were composed by several flows, associated with different kinds of applications, therefore they encompass a realistic mix of both mainly textual streams and binary streams. TCP connections have been reassembled by using TCPflows and the resulting streams have been concatenated to obtain the overall traces.

Dataset	standard DFA (θ=1)	sampled DFA			reverse DFA
		θ=2	θ=3	θ=4	
Snort	3839	4705	4922	4929	784
CiscoA	2431	2683	2627	2497	630
CiscoB	4314	4581	4482	4198	1492

Table I
Number of states for the different automata.

A general concern when working with regex sets is that corresponding DFAs may require a large amount of memory. The sampling operator basically has two opposite effects on the number of states and thus on memory requirements for the resulting sampled DFA: it multiplies the number of transitions for each state by a factor of θ, thus enlarging the number of states too, while reducing the DFA graph diameter (and the overall number of states). While the first consequence is, of course, a side effect, it is however largely compensated by the second one, which keeps the number of states to the same order of magnitude of the original un-sampled DFA. Notice that the number of states corresponding to *Kleene closures* (i.e. $.*$ tokens) is not affected by the adoption of sampling. Tab. I, where also the number of states in the reverse DFAs are shown, proves that our sampling approach does not cost much in terms of memory: generally we need much less than twice the number of states required for an un-sampled DFA.

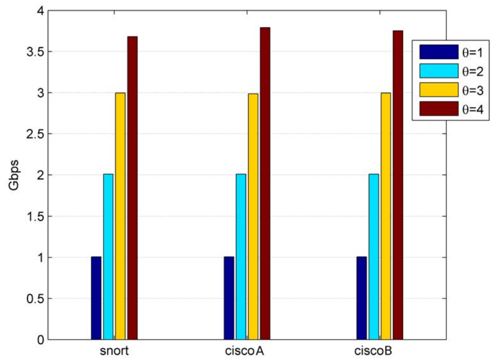

Figure 2. Bit rate with a standard DFA ($\theta = 1$) and sampled DFAs.

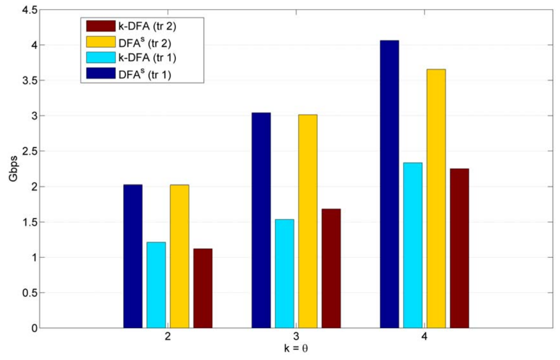

Figure 3. Bit rate with k-DFA and our DFA^s.

However, sampling is orthogonal with any memory saving algorithm, therefore size is not an issue.

To compare our sampling scheme with classical DFA techniques, we used the Network Processor Intel IXP2800 [14] as reference hardware platform. It is fully programmable and characterized by a hierarchy of processors and memories.

We implemented our algorithm by storing the automata in DRAM and reserving for pattern matching 14 microengines (with zero-overhead full threading support). In terms of memory accesses, we measured an average cost of slightly more than a thousands clock cycles for each DRAM access. Fig. 2 reports the results in terms of bit rate when processing a real trace of 20MB with a common DFA (θ=1) and with our sampling scheme (θ =2,3,4). The speedup of the sampled DFA is clear: it multiplies the bit-rate almost linearly by a factor of θ. It is worth noticing that false alarms prevent the bit-rate to grow exactly linearly ($\theta \times$DFA speed) at high θ, but for θ=2,3 the processing required by false alarms is quite negligible. Nonetheless, even with this effect, for θ=4 the bit-rate could be multiplied by a factor of more than $3.5\times$ in our tests.

To compare our solution and the more efficient schemes for speeding up the matching search in DFAs, we implemented the techniques proposed in [6]. In details, we apply alphabet-reduction and default transition compression to k-DFAs introduced in [6], which in turns are based on D^2FAs.

The graph in figure 3 shows the data rates achieved when processing real traces ($tr1$ and $tr2$) and by adopting CiscoB as regex dataset. In particular, we compared our scheme and the one implemented according to the directions in [6] by setting θ=k, where the former represents the sampling period and the latter the stride length (i.e., the amount of bytes which are processed at each step in [6]). Notice that the runs pointed out that both the schemes have no false negatives: in each case (i.e., for each mix of traces and databases) the overall number of signature-matches is detected. The 1^{st} and 3^{rd} bars from the left of the histogram represent the bit-rates for our sampling scheme, while the other two bars report the values for the multi-stride scheme. The advantages of our schemes are clear, with speedup gains between 60% and 95% w.r.t. the multi-stride scheme and 350% over a standard DFA.

VII. Conclusions

Regular expression matching is a widely investigated field that has received a great attention in the last years. However, only few works have proposed solutions to accelerate matching engines by a factor N, by enlarging the amount of bytes to process per cycle. In this paper we have proposed a novel approach that brings the *sampling* idea to the regular expression field. Moved by the motivation that the largest portion of traffic does not match signatures of real databases, we show that sampling the text is an effective technique to speed up the common case. Whenever a match in the sampled text occurs, a reverse DFA is used to confirm the match by observing every character of the text. Results show that the sampling approach is between 60% and 95% faster than previous advanced solutions and 350% faster than standard DFAs, thus proving valuable for implementation in real devices.

References

[1] R. Sommer and V. Paxson, “Enhancing byte-level network intrusion detection signatures with context,” in *Proc. of CCS ’03*. ACM.
[2] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, “Algorithms to accelerate multiple regular expressions matching for deep packet inspection,” in *Proc. of SIGCOMM ’06*. ACM.
[3] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, “Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia,” in *Proc. of ANCS ’07*. ACM, pp. 155–164.
[4] M. Becchi and P. Crowley, “An improved algorithm to accelerate regular expression evaluation,” in *Proc. of ANCS ’07*, 2007, pp. 145–154.
[5] R. Smith, C. Estan, S. Jha, and S. Kong, “Deflating the big bang: fast and scalable deep packet inspection with extended finite automata,” *SIGCOMM CCR*, 2008.
[6] M. Becchi and P. Crowley, “Efficient regular expression evaluation: theory to practice,” in *ANCS ’08*, 2008.
[7] B. C. Brodie, D. E. Taylor, and R. K. Cytron, “A scalable architecture for high-throughput regular-expression pattern matching,” in *ISCA’06*.
[8] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, “An improved DFA for fast regular expression matching,” *ACM SIGCOMM CCR*, October 2008.
[9] G. Khanna, I. Laguna, F. Arshad, and S. Bagchi, “Stateful detection in high throughput distributed systems,” in *IEEE SRDS 2007*.
[10] J. E. Hopcroft and J. D. Ullman, *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman, 1990.
[11] E. K. Ngassam, B. W. Watson, and D. G. Kourie, “Hardcoding finite state automata processing,” in *SAICSIT ’03*, 2003.
[12] J. W. Will Eatherton, *An encoded version of reg-ex database from cisco systems provided for research purposes*.
[13] *Michela Becchi, regex tool, http://regex.wustl.edu/*.
[14] M. Adiletta, et al., “The next generation of intel ixp network processors,” *Intel Technology Journal*, vol. 6, pp. 6–18, August 2002.