Scaling Regular Expression Matching Performance in Parallel Systems through Sampling Techniques

Domenico Ficara
Cisco Systems International Sarl
Avenue des Uttins 5
CH-1180 Rolle (Switzerland)
Email: dficara@cisco.com

Gianni Antichi
Nicola Bonelli
Andrea Di Pietro
Stefano Giordano
Gregorio Procissi

Fabio Vitucci
W.I.N.
Viale R. Piaggio 32 - Pontedera (Italy)
Email: fabio.vitucci@winmed.it

Dept. of Information Engineering
University of Pisa
Via G. Caruso 56122 - 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. For this purpose, regular expressions are used, due to their high expressive power, and Deterministic Finite Automata (DFAs) are adopted to match them. Many works have been proposed to improve DFAs, especially in terms of memory consumption and speed. Instead, we address another issue: the scalability of DFAs to parallel systems and their buffer requirements. To our knowledge, a single attempt to parallelize DFA walk on regular multicore systems (which exploits speculation with limited efficiency) has been proposed in literature. We propose a solution in which a number of processing units are committed to walk in parallel a DFA for the same packet; at this aim, sampling techniques on both text and regular expressions are adopted. This scheme is the first in literature that proposes effective parallelization of DFA walk, hence allowing for packet processing time reduction and less memory for reordering buffers. The result is that speed scales as the number of processing units.**

***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 (IDSs), which commonly use, for deep packet inspection, multiple-string matching algorithms. Instead nowadays, well known tools, such as Snort and Bro, and network devices, such as those of Cisco, have adopted the regular expressions (regexes) to represent string set, due their increased expressiveness [1].

Typically, DFAs are employed for regex matching. They require one state traversal per character only, but an excessive amount of memory. Thus, a large number of works have been presented to solve this problem ([2][3][4][5][6][7]).

Instead, other solutions try to increase the speed of DFA walk, which is still limited by the processing required by every single byte to be observed ([8][9]).

In this paper, we address other issues which arise in real devices using DFAs:

- scalability in parallel systems (this work is one of the first attempts to parallelize DFA-walks);
- length of reordering buffers (DFA walk is expensive as it depends on packet length, and preserving packet order often requires the storage of a high number of packets).

Our idea for relaxing these constraints is “sampling” text: a certain number of execution units (equal to the sampling period) process the same packet with a different starting offset, thus giving partial results which are then collected by a synchronization unit for a final decision. This way, the analysis of each packet is speeded up and the memory required to preserve packet order is remarkably reduced. Obviously, consistency between standard and sampled walk must be ensured: our solution is based on “sampling” the regexes also and building a standard DFA according to these new rules.

The remainder of the paper is organized as follows. In sec. II related works about DFAs are discussed, while sec. III presents the motivations of our proposal. Then, sec. IV illustrates the overall proposed scheme while sec. V shows the experimental results. Finally, sec. VI concludes the paper with some final remarks.

II. Related works

As above mentioned, modern intrusion detection systems base their pattern matching process on regexes, which provide high expressive power and flexibility in string representation [1]. Traditionally, in order to search for regexes, DFAs and NFAs (Non deterministic Finite Automata) are used. 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 regular expressions. Therefore, many works have been proposed focusing on memory reduction of DFAs, in order to make feasible their implementation in real systems. It is important to remark that all these techniques can be adopted in conjunction with the scheme we propose in order to save memory.

Kumar et al. [2] observed that many states in DFAs have similar sets of outgoing transitions which can be replaced with a single default one. However, their idea entails a memory bandwidth increase, which is limited in [4] by exploiting the remark that all regex evaluations begin at a single starting state, and the vast majority of transitions among states lead back either to the starting state or its near neighbors.

In [6] we show that the memory compression of D^{2}FAs can be obtained also with a single memory access per character: our δFAs (Differential encoded Finite Automata) exploit the redundancy of DFAs, where many transitions for a given character are directed to a single state.

Finally, [3] and [5] were the first works to attack the state *blow-up*, that occurs when several distinct regular expressions are merged in a single DFA. They "create" new automata (eXtended Finite Automata and History-based Finite Automata) with an "extended memory".

The second main issue for regex searching techniques is speed, and some works have explored the possibility to increase it. Basically, the idea of these 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 [9] suggest a solution by observing that in actual FAs the number of different transitions (even when k bytes are processed) is more limited. However, their approach is not feasible in contexts where big DFAs (more than 100 states) and/or large compressed alphabets are involved. Therefore, in [8] alphabet-reduction and default transition compression are exploited.

However, while these works discuss ways to accelerate DFA walk in sequential systems, modern network systems are *massively* parallel. Very few proposals try to exploit the potential of parallelism in FA walks: usually walk is simply replicated on different units, each working on a different packet in a run-to-completion mode.

The work in [10] proposes a distributed IDS dividing the load across multiple sensors. Traffic is sliced at frame boundaries, and each slice is analyzed by a subset of the sensors. However, as this method assumes individual, distinct network sensors, it does not directly apply to our topic.

Some parallel algorithms for regexes and string matching have been developed and studied outside of the intrusion detection context. The work in [11] shows that n characters can be matched in $O(\log(n))$ steps given $n \times a$ processors, where a is the alphabet size. This algorithm handles arbitrary regexes but was intended for Connection Machines-style architectures with massive numbers of available processors. Similarly, Misra [12] derives an $O(\log(n))$-time string matching algorithm using $O(n \times \text{length}(string))$ processors. As with the above, the resulting algorithm requires a large number of processors.

The closest previous work [13] breaks the packet in several chunks, scans them in parallel speculating on the result and, if the speculation is wrong, corrects it later. It is based on the observation that, although the actual DFA can have numerous states, only a small fraction of them are visited often while parsing benign traffic. Therefore the method guesses the initial state for all but the first chunk, and then makes sure that this guess does not lead to incorrect results. It is worth noticing that the work in [13] is the only one which targets regular multicore systems and hence can be effectively deployed on nowadays' machines. Nonetheless, the improvement over regular sequential systems is quite limited: measurements show that, by breaking the input into two chunks processed by two cores, this algorithm can achieve up to 24% improvement over the traditional single-core matching procedure.

In this paper we want to propose an innovative technique to parallelize DFA walks which reduces the remarkable amount of memory required to store packets and preserve their order. To this aim, our idea is sampling the text with different engines, thus obtaining partial results to be then used for a conclusive decision. While the idea of sampling is borrowed from [14], here the algorithms are radically different as we consider multicore platforms to take full advantage computation parallelism. The overall result is that rate scales with the number of processing units (hence 100% improvement) and a much smaller reordering buffer is needed.

III. Motivations and main idea

Modern network platforms employ multiple parallel units to scale performance. It is therefore crucial for network applications to adapt to parallel systems in the most appropriate way. Traditionally in applications that perform DFA walk on packets, because of the implicit sequentiality of the operations in, pipelined approaches are discarded and a simple run-to-completion model is adopted, in which each execution unit (*EU*, say a set of threads – *th*) walks a different packet (*per–packet* parallelism). The drawback of this solution is that it may require potentially large buffer memory, especially because of the wide variety of packet sizes in real traffic: since execution time grows linearly with packet size, threads may complete packets in arbitrary order, thus requiring a reordering buffer.

The idea behind this paper is, instead, to take advantage of *in–packet* parallelism through sampling. Sampling is an effective way of reducing service time for a packet: basically, we make P threads (P=sampling period) access the same packet in a sampled fashion (reading a byte every P bytes). Each thread is given a different starting offset, so that P threads can cover all the bytes of a packet. This way, we split the workload of a packet into P smaller workloads to be executed by P different threads.

This approach reduces the need for a reordering buffer (as the group of threads is processing one packet at a time) and, in addition, shows increased memory access locality. Indeed, compared to a standard run-to-completion mode, this scheme works on P-times less packets in P-times less time: this means that data-locality is increased and packets will most likely spend all their execution time in L1 and L2 caches.

We point out that our system is made up of two different stages, a first stage (the working threads) that processes all of the incoming traffic, and a second one (the `sync-thread` which is activated when a matching has to be confirmed), which only examines the traffic which has been flagged as suspicious by the first stage and that would hopefully inspect by walking through an unsampled DFA a small portion of the incoming data only (as the great majority of packets normally belongs to legitimate flows). Such a decomposition is analogous to that shown in [3], and may similarly suffer from a performance degradation when a large amount of malicious traffic is to be processed; such a weakness may be leveraged by an attacker that sends a large amount of malicious-looking traffic in order to overload the intrusion detection system. However, in order to address such an issue, the same solution proposed in [3] can be adopted, which leverages an anomaly index to offer different service rates to anomalous and regular flows.

IV. The algorithm

As just mentioned, the scheme works by allocating P threads to a packet. We refer to such a set of threads as *execution unit* or EU. Each thread of an EU reads bytes from the same packet through sampling, starting from a different offset (hence all payload bytes are accessed), and works on a sampled DFA (see [14]). Notice that we may have different EUs in a parallel system processing different packets.

Our aim is to make these *worker* threads walk through all packet and check for partial matches, even with "false-alarms", leaving the confirmation to a `sync-thread`. Such a `sync-thread` adopts an unsampled DFA which scans the packet byte per byte and hence can confirm the match. It is clear that, since each thread *sees* only a sampled subset of the total payload, a match reported by a thread does not imply a real match in the text.

As intuitive, false alarms and confirmation stages may slow down the overall process, thus we design our scheme so that false alarms are reduced and the `sync-thread` works with small text intervals. For this reason, we describe regexes as a sequence of closure-separated substrings of the form: $\boldsymbol{r_1}.*\boldsymbol{r_2}.*\boldsymbol{r_3}\ \ldots$It is evident from this notation that matching a regex corresponds mainly to matching its sequence of substrings $\boldsymbol{r_1, r_2, r_3 \ldots}$

Each closure in the previous notation introduces at least a *root-state* in the corresponding DFA. Indeed, we define as root states the states in the DFA with a closure (at least one transition pointing to the state itself). These states are the only ones which allow the automaton to stay steady (i.e. to process the incoming characters without changing its internal state) when no further matching substring is being detected. Hence, in order to reduce the text interval to be confirmed by the `sync-thread`, we let worker threads of an EU match substrings between root-states.

The details of the algorithm to be executed by each thread are illustrated in alg. 1. Notice that $Nmatch$, $partial\_matching$, N_{moving} and $begin\_sync$ are global variables shared by all of the threads (this is the reason why they are handled within a critical section).

In alg. 2 we show the pseudocode of the `sync-thread`, which walks through the standard DFA to confirm the correctness of substring matching in the text-interval indicated by worker-threads. Note that the additive cost of `sync-thread` is negligible (as it will be shown in sec. V).

The choice of P affects many parameters: the higher the value of P, the lower the number of bytes to allocate for reordering buffer, and the lower (at most, equal) the time a thread can be in idle state (and consequently the faster the overall traffic processing). However, the value of P is limited by substrings length (as shown in [14], P must be smaller than the smaller string to match); moreover, a high value of P reduces the calls to the `sync-thread`, as it is less likely that many threads detect a matching substring. On the other hand, a large period P may increase the number of false positives (which lead to `sync-thread` calls). These effects will be evaluated in sec. V.

Algorithm 1 Pseudo-code for the sampling procedure

```
procedure worker_thread(P,$D^s$)
1:  while (true) do
2:      $c \leftarrow getchar(pos)$ /*$c$: character at current position $pos$ */
3:      $s_{next} \leftarrow D^s.walk(c, s)$ /*Note: $D^s$ is Sampled DFA */
4:      /* $s$: current DFA state, $S_{next}$: next state to visit on DFA. */
5:      enter critical section
6:      if !$s_{next}.is\_matching$ then
7:          $Nmatch++$
8:          if $Nmatch = P$ then
8:                  SAMPLED_MATCH, Wake Sync-thread
9:      if $s_{next} \neq s$ then
10:         $n_{moving}++$
11:         if $n_{moving} = P$ then
12:             $matching\_in\_progress \leftarrow true$
13:     else
14:         if $matching\_in\_progress$ then
15:             $matching\_in\_progress \leftarrow true$
16:             SAMPLED_MATCH_OVER, Wake Sync-thread
17:         $begin \leftarrow current\_char\_pos$
18:         $n_{moving} \leftarrow 0$
19:     exit critical section
```

Algorithm 2 Pseudo-code for synchronization

```
procedure sync_thread(D)
1:  while (true) do
2:      while !$sampled\_match$ do
3:      for $i \leftarrow begin, i < end$ do
4:          $s \leftarrow D.walk(c, s)$ /* Note: $D$ is Unsampled DFA */
6:          if $s.accepting$ then
7:              $MATCH!$
```

V. Algorithm Evaluation

The evaluation of the proposed algorithm is carried out by both simulations and actual runs over an experimentally developed prototype. Next subsections elaborate upon both approaches in details.

	Cisco			Snort		
Period	1	2	4	1	2	4
Max Buffer Size	296 KB	121 KB	47 KB	294 KB	120 KB	51 KB
Overall Data Rate	2.21 Gbps	2.13 Gbps	2.06 Gbps	2.23 Gbps	2.17 Gbps	2.09Gbps
L1 Cache hit-ratio	82.3%	81.4%	81.1%	94.6%	92.3%	92.2%
L2 Cache hit-ratio	72.0%	79.2%	81.2%	33.4%	90.8%	94.1%
Sync-Thread Calls/Byte	–	1.2×10^{-2}	5×10^{-4}	–	1.1×10^{-2}	6.1×10^{-4}

Table I
RESULTS OF SIMULATIONS ON A TRACE OF 500MBYTES.

A. Simulation Assessment

We first present the evaluation of our scheme in an environment simulating modern multi-core architectures. More precisely, in the simulated processor i) each core supports a number of physical threads and has its own L1 data cache, ii) cores are grouped in clusters which share the same L2 data cache, iii) for simplicity, threads are executed in a round robin fashion within the same core. In details, we simulate a 4-core CPU with 2 physical threads and a 128KB L1 data cache for each core; a single 4MB L2 data-cache is also available (thus reproducing for example an Intel XEON E5365 CPU). Simulations are performed on traces of over 500MB taken from the router of our department laboratory.

In fig. 1 a histogram of the amount of bytes on the reordering buffer is depicted for a regular run-to-completion unsampled system and for a sampled system with $P = 2$ and $P = 4$. The figure clearly shows that, as P grows, the distribution of the bytes occupied in the reordering buffer gets more concentrated around lower values, thus confirming our assumptions. The histogram also confirms the intuition that, if a buffer size of B is needed in the unsampled case, B/P bytes are needed in the buffer in the sampled case.

Table I shows the maximum buffer size, packet rate, `sync-thread` calls and L1/L2 cache hit ratios for two sets of regular expressions: the Cisco dataset (courtesy of John Williams and Will Eatherton) and a Snort subset of signatures. Both sets include over a hundred regular expressions which produce a DFA of over 10k states each. We already mentioned that our technique aims at reaching a trade–off between computation speed–up (i.e. speed multiplication obtained by using more threads) and reordering buffer size. In order to evaluate such a trade–off we compared both of these performance indexes to those measured with a naive scheme, where packets are distributed across the threads and each thread performs standard DFA processing (in the table, this case is referred to as $P = 1$, meaning unary sampling period). Of course, we expect the throughput of our technique to be lower due to the overhead of matching confirmation performed by the `sync-thread`. The table confirms our intuition that sampling is a cache-friendly technique: L2 cache hit-ratio grows with P, while L1 hit-ratio decreases very slightly but generally remains stable. Basically, this small reduction in L1 hit-ratio may be explained by the communications with `sync-thread` which involves data-sharing and write-operations, as opposed to pure read-operations which occurs in regular DFA-walks. Both the slight L1 cache hit ratio reduction and the `sync-thread` operations slightly reduce the overall operations, thus affecting data rate, which is not exactly the same in the sampled case as in a regular unsampled system. However, the differences are minimal and the advantages of having a reduced reordering buffer often outweigh this problem. Compared to the work in [13] where two cores allow for a 24% rate improvement, our scheme provides almost 100% rate increase. Calls to `sync-thread` are also very rare, in the order of one call every hundred of payload bytes in the worst case. Moreover, while we may expect a larger number of false alarms occurs as P grows (because sampling reduces regex length and enlarges the set of matching text), the opposite happens. As already mentioned in section IV, this effect may be explained by the operations performed by worker-threads: as P grows, in order to call the `sync-thread` more threads need to reach the same root-state within a period, which gets, of course, more difficult.

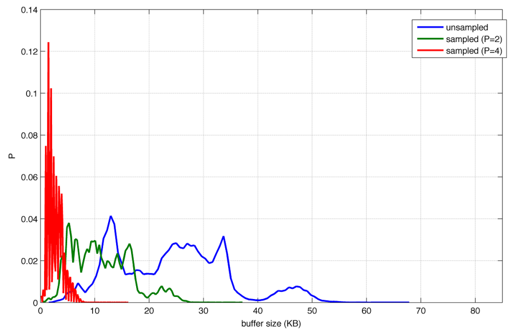

Figure 1. Histogram of Buffer size (Bytes)

B. Experimental Assessment

In order to evaluate the algorithm in a more realistic scenario, we also implemented a real prototype of our algorithm and we make it process real traffic collected over a local network. For fair comparison, we also implemented a prototype of a standard parallel packet inspection application using regular unsampled DFAs which distribute the packets across the processors in a FCFS manner. We ran both applications on a commodity PC hosting an Intel i7 processor providing 4 cores and 8 GBytes of RAM memory and running a Linux Debian distribution. In all experiments, we read traffic from

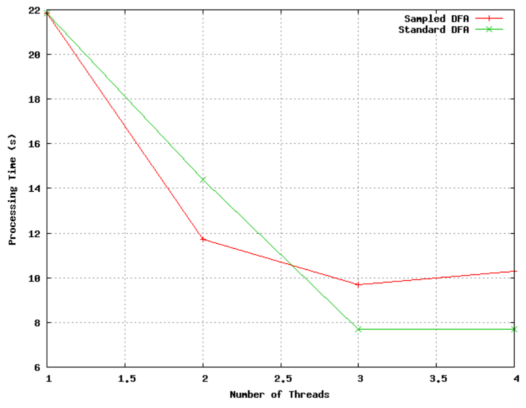

Figure 2. Snort Ruleset

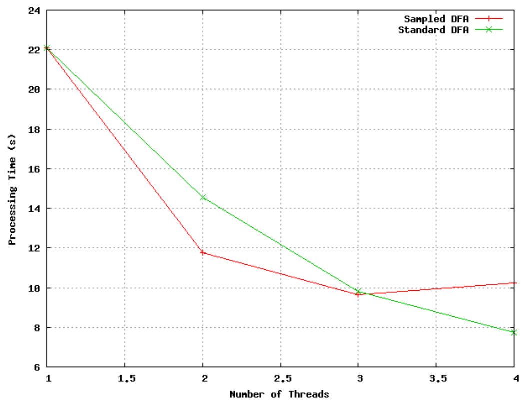

Figure 3. Cisco Ruleset

a 500MB long *pcap* trace collected from a local network and stored on the local hard drive. Figures 2 and 3 show the overall processing times needed to inspect the whole trace by the two algorithms when using the same Snort and Cisco rulesets adopted in the simulation approach. Notice first that, as in our algorithm the number of threads is equal to the sampling period P, in the single thread case the two algorithms collapse, so do their performance, obviously. As expected, the standard parallelized application exhibits smaller completion times as the number of cores increase. This is due, as already pointed out, to the additional processing performed by the `sync-thread` as well as to the more complex synchronization paradigm (not only mutual exclusion but also strict ordering has to be guaranteed). Surprisingly, though, our scheme achieves better results when only *two threads* are used: the reason for that is probably again the synchronization paradigm (the ordered approach seems to be more efficient with two threads only). Overall, the experimental results confirm that our approach achieves a speed–up which is comparable to that of per-packet parallelism, while considerably reducing the need for output buffering.

VI. Conclusions

In this work we propose a technique to exploit parallel systems thus allowing for an efficient scaling in modern network systems, also reducing buffer requirements. We propose DFA walk parallelization through sampling techniques: an execution unit composed by a certain number (P) of processing units takes care of the same packet sampling its payload and a final `sync-thread` collects the partial matchings confirming the global matching. Hence, the time required for each packet and consequently the reordering buffer length are remarkably reduced, with good scaling capabilities as the number of cores increases. This work is the first in literature to achieve efficient results with regular multicore systems.

Results confirm the effectiveness of the proposed approach: required buffer size reduces by a factor of P with respect to the standard case, with a negligible decrease in packet rate.

Acknowledgements

This work was partially supported by DEMONS, a research project supported by the European Commission under its 7th Framework Program (contract-no. 257315). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DEMONS project or the European Commission.

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 Comput. Commun. Rev., 2008.
[6] 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.
[7] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. D. Pietro, “Faster dfas through simple and efficient inverse homomorphisms,” in Proc. of INFOCOM ’09.
[8] M. Becchi and P. Crowley, “Efficient regular expression evaluation: theory to practice,” in ANCS ’08, 2008.
[9] B. C. Brodie, D. E. Taylor, and R. K. Cytron, “A scalable architecture for high-throughput regular-expression pattern matching,” in Proc. of ISCA’06, June 2006.
[10] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer, “Stateful intrusion detection for high-speed networks,” in Proc. of SP ’02. IEEE Computer Society.
[11] W. D. Hillis and G. L. Steele, Jr., “Data parallel algorithms,” Commun. ACM, vol. 29, no. 12, pp. 1170–1183, 1986.
[12] J. Misra, “Derivation of a parallel string matching algorithm,” Information Processing Letters, vol. 85, pp. 255–260, 2001.
[13] D. Luchaup, R. Smith, C. Estan, and S. Jha, “Multi-byte regular expression matching with speculation,” in RAID, 2009, pp. 284–303.
[14] D. Ficara, G. Antichi, A. Di Pietro, S. Giordano, G. Procissi, and F. Vitucci, “Sampling techniques to accelerate pattern matching in network intrusion detection systems,” in Communications (ICC), 2010 IEEE International Conference on, 23-27 2010, pp. 1–5.