Scalable Pattern-Matching via Dynamic Differentiated Distributed Detection (D^4)

Kai Zheng[1], Hongbin Lu[1,2]

[1] IBM China Research Lab [2] Department of Computer Science, Tsinghua University, Beijing, China

zhengkai@cn.ibm.com, lu-hb02@mails.thu.edu.cn

Abstract—**Pattern Matching (PM) over network packet flows for Network Intrusion Detection/Prevention System is becoming more and more performance sensitive due to the rapid progress of Internet applications in terms of data volumes. Meanwhile, modern multicore platforms are becoming performance competitive with traditional hardware solutions for PM. But due to the unbalance of network flow sizes, traditional flow-based data parallel processing/programming model can not fully exert multicore platforms' computing power and results in poor performance scalability. In this paper, a novel parallel inspection model, Dynamic Differentiated Distributed Detection (D^4) is proposed. D^4 deploys distributed parallel operations by adding one more dimension on workload partition/allocation. It proposes an effective and efficient scheme to pre-partition the pattern set in several candidate ways, called "Detection Modes", and let multiple candidate PM methods to handle the subsets, respectively; the most suitable *Detection Mode* would be selected specifically for each incoming flows at the run-time, and the workload would be dynamically allocated among multiple CPU cores. Experimental results on real-world pattern set and traffic traces show that D^4 scales much better than traditional schemes by better balancing the load among the processors while avoiding unnecessary overheads.**

Keywords-pattern matching, multicore, distributed, load balancing, deep packet inspection, intrusion detection, parallel

I. INTRODUCTION

Pattern Matching (*PM*) against network traffic is a key enabling technology for many emerging/popular-gaining network applications such as *Network Intrusion Detection/Prevention System* (NID/PS) to detect/prevent malicious intrusions which incur hundreds of millions of lost every year[1]. PM is well recognized as a kind of unwieldy workload to handle since it is both computation and communication (i.e. I/O) intensive in its nature[2].

The traditional hardware-based solutions leveraging FPGA [4-5] or ASIC [6-9] techniques, which always require dedicatedly designed components, are thought to offer relatively higher performance than the software-based ones. However, hardware approaches are always with high price, long time-to-market and low adaptability/flexibility, all of which prevent such approaches from wide deployment.

In comparison with the hardware approaches, the open source IDS software forums and products, e.g. SNORT[13] and Bro_IDS, etc., provide us a much more publicly available, flexible and economical way to realize a solid worldwide anti-attack "alliance"; meanwhile, the modern general purpose multicore technology provides a promising platforms to host the open source NID/PS products. Note that modern homogeneous/heterogeneous multicore processors, such as Intel's Clovertown, AMD's Barcelona, Sun's UltraSPARC T1, and IBM's CELL, POWER 6, outperform the last generation of *Network Processors* in many aspects besides higher clock rate.

However, it is not a trivial work for the open source products to fully exert the power of the modern multi-core system. Note that traditional software parallel programming models for load balancing is inefficient, since the network contents are usually in forms of packet flows/sessions, and the packets within a flow are with strong data dependency with each others and must be processed in sequence to avoid false negatives. An unfortunate fact is that the sizes of network flows might be extremely uneven, and some flows may even dominate most of the cable bandwidth for a certain time interval in specific cases.

All signs lead to the demand on developing a new programming/processing model for high performance PM over the multi-core platforms for network security applications. Such a model needs not only to be with the instinct of retaining data dependency, but also to balance the workload among the parallel processing resources, in dynamic ways.

II. DESIGN CONCEPTS

A. Distributed Detection (D^2) vs. Variation of Flow Sizes

As is mentioned before, patterns indicating malicious attacks may be intentionally separated and spread among different packets by attackers to evade being locked down. To avoid false negatives, NID/PSes have to be with the ability to reassemble the packets and reconstruct the corresponding sessions of the communications before detection. For example, *SNORT* [13] uses several preprocessor plug-ins to defrag IP fragments and to rebuild the TCP flows/sessions before deploying PM on the network data.

Note that, on the other hand, it is an obvious fact that network flows varies largely in terms of size and duration time, and size distinction up to several orders of magnitudes can be observed usually[14]. As depicted in Figure 1, traditional flow-based load-balancing scheme may lead to unbalanced load shares and low utilization of the multiple cores' computing power, resulting in poor scalability.

The primitive idea of *Distributed Detection* (D^2) comes from a key observation on the nature of PM that, different from many other computation intensive workloads, PM handles two kinds of user data, the network strings in terms of packet flows and rules/patterns to be matched against.

978-1-4244-2324-8/08/$25.00 © 2008 IEEE.

Therefore, besides partitioning the workload via packet flows, one may also partition the workload via partitioning the pattern set, as shown in Figure 2.

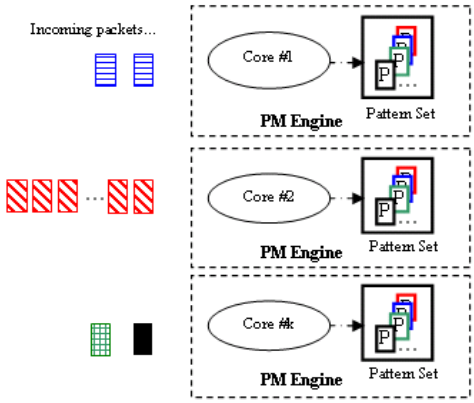

Figure 1. Traditional Flow-based Load-Balancing. (Different colors denotes different flows)

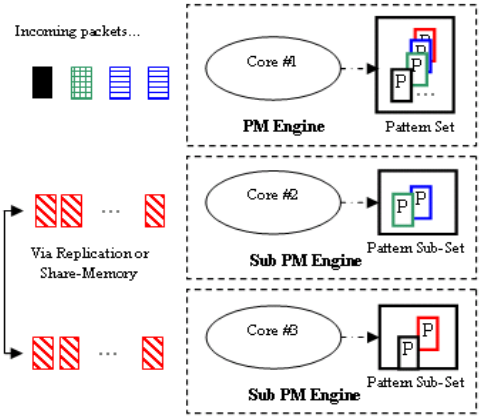

Figure 2. Reallocating/Balancing the workload via D^2.

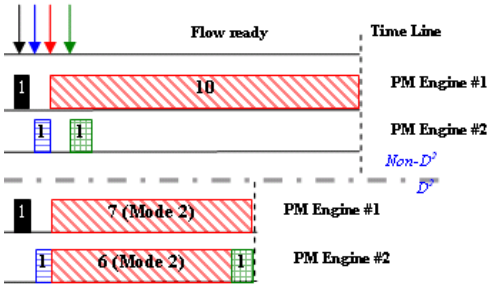

Figure 3. A positive case for D^2, in which Mode 2 is applied to the hot (red-color downward diagonal) flow when both PM Engines are idle. D^2 brings gains by achieving a better CPU utilization, despite the overhead introduced (6ms+7ms>10ms).

Formally, for a specific packet flow to be inspected, we define the *Detection Mode* as *Mode M*, when *M* processors/cores work together on the given flow, and each of them handles one of the *M*-partitioned pattern subsets, respectively. In the case shown in Figure 2, the small flows denoted in black, green (grid) and blue (horizontal line) are allocated to core#1 and detected in Mode 1, while the huge flows denoted in red (downward diagonal) are detected in Mode 2 that both core#2 and core#3 will together inspect the flow, each of which against one of the 2-way-partitioned subsets, respectively.

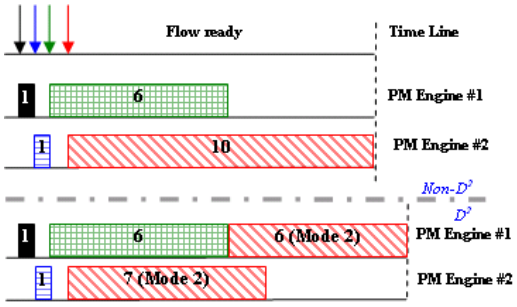

Figure 4. A negative case for D^2, in which Mode 2 is applied to the hot (red-color) flow when PM Engine#1 is busy.

B. Dynamic D^2 to Avoid Unnecessary Overhead

On the other hand, note that, detection modes other than Mode 1 may incur uncertain overhead, e.g. 1) from the OS/system, for increased number of memory references to address the data structures of the subsets, or 2) from the algorithmic aspect, for less room for optimization. The higher mode used, the higher overhead may be required (due to page limitation, please also refer to [12] for detailed analysis of the overhead). According to such an observation, detection modes should be carefully chosen in the runtime. The cases in Figures 3&4 illustrate the gain benefited from D^2, as well as the potential risk when taking D^2 unconditionally. In the case of Figure 3, since PME#2 is pretty idle, offloading "parts of" PME#1's task to PME#2 results in a win of the overall performance, despite the overhead in terms of more CPU cycles/instructions. In the case of Figure 4, the hot flow arrives when another large flow (green grid) is being processed by PME#1. In such a case, applying Mode 2 to the hot flow would, on the contrary, decrease the overall performance due to the overhead introduced.

Generally speaking, large mode would be only feasible for the flows which are obviously huger than the ordinary ones, when idle resources are reported.

C. Differentiated D^2 to Develop Algorithmic Gains

One might ask, "Does D^2 always result in net processing overhead and bring great risk of decreasing the overall performance?" According to our research, the answer can be negative. Note that, actually, D^2 is rather a programming framework/model for PM on the multicore platforms than a specific PM algorithm. Under such a framework, one or more PM algorithms can be involved to handle different parts of a pattern set. As we know that for most PM algorithms, their practical performances relate to the characteristics of the pattern set to be matched against. By dedicatedly selecting the rule set partition method and choosing the suitable algorithms to handle each of the subsets respectively, performance gain rather than net overhead can be achieved.

For example, the Modified-Wu-Manber (MWM) algorithm [3] and the Aho-Corasick (AC) algorithm [11] are the two key PM methods adopted by SNORT v2.x. MWM is known to be with a preferable average matching performance

and relatively much lower memory requirement, but it would not be handy and its performance becomes non-deterministic when dealing with short patterns (since the Bad-Character shifts are bounded by the minimum pattern length of the set) and when hash collisions occur heavily. In contrast, from the algorithm's perspective, the AC algorithm is thought to be able to deliver a better performance in the worst-case scenario and a much more deterministic performance which is nearly nothing to do with characteristics of the pattern set; however, besides having a relatively lower average performance, the AC algorithm always consumes much more memory, especially when dealing with huge pattern sets (e.g. AC usually consumes 1~2 orders of magnitude more memory than MWM).

Under the Differentiated D^2 framework, we may try to partition the rule set according to the strengths and weaknesses of the algorithms and let them handle the subsets they are "adept at". For instance, the short patterns are put into a separated subset which is handled by AC, and the remaining patterns, which are supposed to be handled by MWM, are then allocated to other subsets aiming at maximizing the Bad-Character Shift value and minimizing the hash collisions of each of them. By these efforts, we may expect performance gain from the D^2 framework.

III. The Prototype of Dynamic Differentiated Distributed Detection (D^4)

A. System Architecture

As depicted in Figure 5, the proposed PM framework mainly consists of a *Flow Buffer*, a *Mode Selector & Scheduler (MSS)*, several *Pattern Matching Engines* (*PMEs*) , and a *Pattern-Set-Partitioner (PSP)*.

PSP is responsible for pattern set partitioning and data structure optimization. It determines how to organize the data structures representing the patterns in the preprocessing stage, which would be further discussed in *Section III.B*. *Flow Buffer* is where the incoming traffic flows are buffered and reassembled. *PME* is the key component which deploys the matching operations. Corresponding to each PME is a task-info queue each item of which stores the information denoting which flow to inspect and which pattern set/sub-set to detect against. *MSS* determines, at the run-time, for a specific packet flow which mode to take, for example Mode *m*, according to the flow sizes and the status/availabilities of all the PMEs, and it will then dynamically select *m* PME(s) out from all (e.g. based on the availabilities/workloads of the PMEs) to inspect the current flow against the *m* pattern set(s)/subset(s), respectively (the allocation info would be stored in the corresponding task-info queue(s)). Further discussion on MSS is in *Section III.C*.

B. Implementation of the Pattern-Set-Partitioner

As mentioned in Section II, in the preprocessing stage, PSP needs not only to divide the pattern set into segments/subsets for distributed detection among multiple PMEs, but also to decide which PM methods be used for a specific subset so as to achieve best expected overall performance.

In the proposed prototype, we involve both AC and MWM algorithm, and implement a 3-stage PSP framework. For the rule set partition of each candidate Mode *m*, $1 < m \le M$ (*M* denotes the number of PMEs, note that there is no need to do anything here for Mode 1, having nothing to partition).

In the first stage, the patterns would be sorted in the increasing order of their lengths, and then the first $n_{ac}(m)$ ones would be taken out to form the first subset which would be handled by the AC algorithm. Here in the prototype, the parameters $n_{ac}(m), 1 < m \le M$, called the *AC_Threshold*, are chosen as the number of patterns with lengths no larger than a give size (the details of such tradeoff will be described in Section IV). The rest of the patterns would be handled by the MWM algorithm and would be partitioned into *m*-1 subsets in the following stages.

If $m \ge 3$ (i.e. there would be no less than 3 subsets), the rest of the patterns would be firstly divided into $\lceil (m-1)/2 \rceil$ interim sets in the second stage. In this stage, the patterns would be separated in order to minimize the hash collisions in the MWM data-structures; in other words, the patterns with the same hash key would be intentionally allocated to different interim sets in this stage.

Then, in the last stage, each interim sets would be partitioned into two subsets aiming at optimizing the Bad-Character Shift (BCS) table of the MWM data structures corresponding to each subset; in the proposed prototype, we

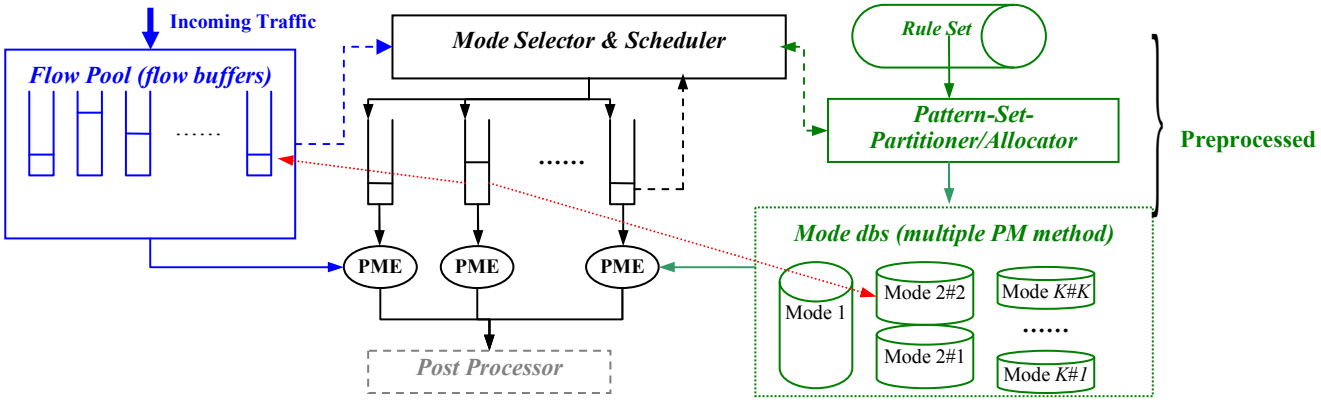

Figure 5. The schematic of the proposed DDD framework.

implemented an *Incremental Pattern Allocation Scheme* (IPAS) to partition the interim set. Two cost counters would be maintained for the two pattern subsets, respectively. Each is equal to the average BCS value of the corresponding BCS table of the MWM data-structure. Patterns are allocated sequentially, and for a specific pattern, firstly two testing assignments would be deployed on both subsets to get the updated cost counter values, respectively; then the pattern would be assigned to the subset incurring less cost increment. Figure 6 shows the pseudo-code of the prototyped PSP scheme.

Input: The original pattern set PS_{orig}, Mode# m, and AC_Threshold for Mode m $n_{ac}(m)$

Output: The m-partitioned pattern subsets $\{ PS_{mode-m}(k), k = 1,..., m \}$

Stage 1:

Sort p_i ($p_i \in PS_{orig}$) in the increasing order of their lengths, resulting in $\{p'_i \mid p'_i \in PS_{orig}, p'_i.length \le p'_j.length, \forall i \le j\}$;

Let $PS_{mode-m}(1) = \{p_i, i \le n_{ac}(m)\}$;

Stage 2:

If m==2 **Then**:

Let $PS_{mode-m}(2) = PS_{orig} - PS_{mode-m}(1)$, **Done**;

Else:

Interim set number $N_{Int} = \lceil (m-1)/2 \rceil$;

For $p'_i \in (PS_{orig} - PS_{mode-m}(1))$

Hash key K =Hash(p_i');

Round-Robin indicator $I[K] = (I[K]+1)\%N_{Int}$;

Assign p'_i to the $I[K]^{th}$ Interim set, $IS_{I[K]}$;

Stage 3:

For k from 1 to N_{Int}

For $p'_i \in IS_k$

V=Get_BCS_avg($PS_{mode-m}(2k) \cap \{p'_i\}$);

U= Get_BCS_avg($PS_{mode-m}(2k+1) \cap \{p'_i\}$);

If $V \le U$ **Then**:

Assign p'_i to $PS_{mode-m}(2k)$;

Else:

Assign p'_i to $PS_{mode-m}(2k+1)$;

Figure 6. The pseudo-code of the prototyped PSP algorithm to partition the original pattern set into m subsets (i.e. for Mode #m).

C. Implementation of Mode Selector & Scheduler

Mode Selector & Scheduler (MSS) stands at the heart of the D^4 framework. As analyzed in Section II, improper choice of the detection modes may, on the contrary, incur performance degradation, due to the overhead introduced. We know, that usually there would be non-neglectable cost for the OS to call/change the PM methods or to switch the pattern subsets (e.g. to load the corresponding codes and the pattern data structure to the cache, which may lead to cache thrashing and CPU stall), unconditionally or frequently doing so may bring un-affordable system level overhead to the overall performance. Therefore, to make the D^4 framework indeed work over the modern multi-core platforms, the following factors need to be taken into consideration in the design.

1) It tends to be always un-worthwhile to apply D^2 (i.e. with Detection Modes other than Mode 1) on small flows, since firstly small flows is easy to be scheduled and would be less possible to incur "out-of-balance" issues, and secondly, the overhead for the small flows would be too large in terms of per byte counts; typically, flows shorter than some tens of KBs should be called as the "small flows" and the ones longer than 1 Mega bytes can be regarded as the larger ones and ready for D^2, according to our experiments across several multicore platforms.

2) If the gain by Differentiated D^2 is always less than the overhead introduced for a specific case, the system may not be always ready for D^2, even for the large flows; note that D^2 provides for the multicore system only the way to gear up its CPU utilization, so as to achieve better scalability when processing unbalance-sized flows; given that the system is already very busy and would remain busy for a while, applying D^2 would merely tire the system out.

3) According to the observations on the numerous experiments done, we found that, even for the same traffic trace and the same pattern sets, the PM performance may vary significantly from system to system when using a given Detection Mode; in other words, MSS should also take account of the characteristics of the system or try to "adapt" to the system, e.g. a pre-test on the system (using certain sample traces) may be necessary when determining the parameters for dynamically mode selecting.

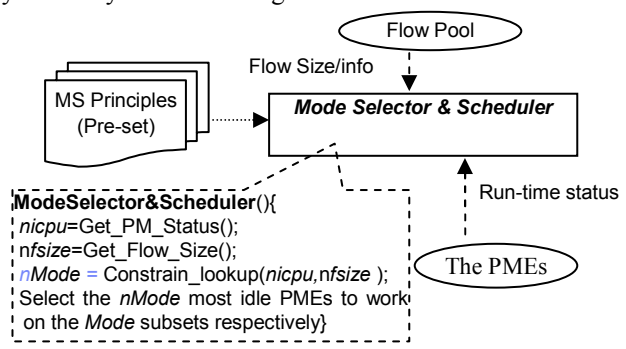

Figure 7. The Schematic of Mode Selector & Scheduler.

Figure 7 illustrates the ecosystem and the data-flow of MSS. In the proposed prototype, we set some simple static rules for MSS. Note that, Detection Mode should be selected for each flow; therefore MSS must be implemented as simple as possible to avoid un-affordable overhead. Experimental results show that, even very simple algorithms/rules for MSS to select detection mode in the real-time can realize pretty good adaptability that idle processor resources are well utilized while avoiding the negative effects.

IV. EVALUATIONS

We evaluated the proposed D^4 framework by conducting several scalability experiments on a SunFire T2000 (Niagara) multicore server platform. The server has an 8-core UltraSPARC T1 processor; each core has 4 "zero-penalty-content-switched" hardware threads to hide processing latency. This features let it act as a SMP platform with 4x8 logic CPUs and very good for the application scalability study.

The pattern set from SNORT[1] and the traffic traces from DEFCON [10] (collected in the real-world[2], which consists of

[1] Dated April 25, 2006, including totally 16571 patterns within 4085 rules

[2] The traces are TCP reassembled in terms of flows, with all headers

100K packet flows and totally 285,341,906 bytes, the largest flow of which is 3,854,144 bytes in size) are used.

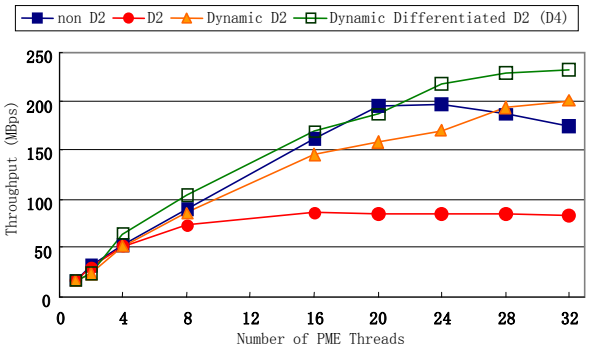

Figure 8. Throughput scalability comparison among different schemes.

Four different MWM-based parallel PM schemes are evaluated, including:

i. The straightforward per-flow-based load balance scheme (i.e. the non-D^2 scheme using Mode 1 merely);

ii. The Brute-force D^2 scheme in which the Detection Modes are equal to the number of PME threads used;

iii. The Dynamic D^2 scheme in which Detection Modes are selected in the runtime according to the criteria shown in Figure 7;

iv. The proposed D^4 scheme, which is similar with the Dynamic D^2 Scheme except that the patterns whose sizes are not larger than 9 bytes would be processed by the AC algorithms when *Mode*>1.

Figure 9 depicts the PM throughputs scaling with the number of PME threads used. As for the non-D^2 scheme, we can see that, it scales well only with less than 20 PME threads; due to the inability to handle the large flows in a parallel way, some PME threads tend to be idle when more threads are getting involved. As for the Brute-force D^2 scheme, very poor scalability is observed, when more than 8 PME threads are used; although a pretty good CPU utilization scalability is achieved, however, quite a lot of the CPU cycles are wasted in processing the overheads. Both the Dynamic D^2 scheme and the D^4 scheme show similar throughput growth trend, which is distinctively better than the previous ones (e.g. D^4 outperform the non- D^2 scheme by up to 30%), except that D^4 achieves a better performance in terms of absolute value, due to the additional gain by leveraging different PM methods and exert their strengths, respectively. (Due to page limitation, please also refer to [12] for more detailed analysis.)

V. CONCLUSIONS

In this paper, a novel parallel programming framework, called *Dynamic Differentiated Distributed Detection* (D^4), is proposed to optimize flow-aware pattern matching on the modern multicore processing platforms. The primitive idea of D^4 comes from the key observation that network data always needs to be processed in terms of flows which usually exhibit un-balanced distribution in sizes, and therefore makes the workload hard to be evenly partitioned among the CPU cores, resulting in a low scalability of the performance. D^4 deploys distributed parallel operations by adding one more dimension on workload partition/allocation. It proposes an effective scheme to pre-partition the pattern set in several candidate ways, and let multiple candidate PM methods (e.g. the ones complementary with each others in performance) to handle the subsets, respectively; the most suitable "Detection Mode" would be selected for any specific incoming flows at the run-time, and the workload would be dynamically allocated among multiple CPU cores, according to the flow sizes, the load of the cores, or other associated factors. Experimental results on the proposed prototype using real-life pattern sets and traffic traces show that D^4 provides a good optimization on cores utilization and achieves distinct performance gain against traditional non-D^2 load-balancing scheme; furthermore, the overhead introduced by distributed detection is perfectly reduced by leveraging some heuristic rules based on run-time information.

removed, and randomly inserted with more "dirty" strings to generate a semi-worst-case scenario, for pressure testing.

VI. REFERENCES

[1] "2005 FBI Computer Crime Survey," http://www.digitalriver.com/v2.0-img/operations/naievigi/site/media/pdf/FBIccs2005.pdf.

[2] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating realistic workloads for network intrusion detection systems," ACM Workshop on Software and Performance, 2004.

[3] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Techinical Report TR-94-17. Department of Computer Science, University of Arizona, 1994.

[4] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2004.

[5] Y. H. Cho and W. H. Mangione-Smith, "Fast reconfiguring Deep Packet Filter for 1+ Gigabit Network," IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), Napa Valley, CA, 2005.

[6] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM " in Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP'04) - Volume 00 IEEE Computer Society, 2004 pp. 174-183

[7] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," IEEE Micro, vol. 24, 2004.

[8] J. v. Lunteren, "High-Performance Pattern-Matching Engine for Intrusion Detection," IEEE INFOCOM 2006, Barcelona, Spain, 2006.

[9] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu, "A Memory-Efficient Parallel String Matching Architecture for High-Speed Intrusion Detection," IEEE Journal on Selected Areas in Communications, vol. 24, pp. pp. 1793-1804, 2006.

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

[11] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," communications of the ACM, vol. 18, pp. 333-340, 1975.

[12] K. Zheng, H.B. Lu, "Scalable Pattern-Matching via Dynamic Differentiated Distributed Detection (D4)", Technical Report on DPI, available on http://s-router.cs.tsinghua.edu.cn/~zhengkai/Paper/DPI_Technical_Report_2007.pdf

[13] "Snort - the de facto standard for intrusion detection/prevention," http://www.snort.org, .

[14] N. Dukkipati and N. McKeown, "Why Flow-Completion Time is the Right Metric for Congestion Control," ACM SIGCOMM Computer Communication Review, vol. 36, pp. 59-62, 2006.