Improving the Performance Efficiency of an IDS by Exploiting Temporal Locality in Network Traffic

Govind Sreekar Shenoy*, Jordi Tubella* and Antonio González*†
*Department of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, Spain.
†Intel Barcelona Research Center, Barcelona, Spain.
Email: {*govind,jordit*}*@ac.upc.edu, antonio.gonzalez@intel.com*

***Abstract*—Network traffic has traditionally exhibited temporal locality in the header field of packets. Such locality is intuitive and is a consequence of the semantics of network protocols. However, in contrast, the locality in the packet payload has not been studied in significant detail.**

In this work we study temporal locality in the packet payload. Temporal locality can also be viewed as redundancy, and we observe significant redundancy in the packet payload. We investigate mechanisms to exploit it in a networking application. We choose Intrusion Detection Systems (IDS) as a case study. An IDS like the popular Snort operates by scanning packet payload for known attack strings. It first builds a Finite State Machine (FSM) from a database of attack strings, and traverses this FSM using bytes from the packet payload. So temporal locality in network traffic provides us an opportunity to accelerate this FSM traversal. Our mechanism dynamically identifies redundant bytes in the packet and skips their redundant FSM traversal. We further parallelize our mechanism by performing the redundancy identification concurrently with stages of Snort packet processing.

IDS are commonly deployed in commodity processors, and we evaluate our mechanism on an Intel Core i3. Our performance study indicates that the length of the redundant chunk is a key factor in performance. We also observe important performance benefits in deploying our redundancy-aware mechanism in the Snort IDS[32].

I. Introduction

The constant evolution of Internet has demanded an increased functionality from the network-layer, the layer in the network stack where routers operate. Network-layer applications like packet forwarding are computationally very intensive. Packets can be operated on independently and, hence packet level parallelism is a unique and a common feature among network-layer applications. Further, this feature is aggressively exploited in network processors with multi-cores and multiple threads. For example, Intel IXP 2400 has eight cores and eight threads per cores. Network-layer applications also exhibit temporal locality in packet headers. This locality is well exploited using application specific caches like in IP-Lookups and packet classification[5, 9, 24].

While locality in header fields is intuitive and well studied, to the best of our knowledge there have not been significant studies on locality in the packet payload. In fact, the first generation of Intel network processors (Intel IXP 2400) only use caches targeted for lookups in packet headers. Recently, however, there have been works[1, 4] investigating temporal locality in the packet payload. They observe significant locality and use it for efficient transmission of data. Temporal locality can also be viewed as redundancy, and they propose mechanisms to compress the transmission of these redundant bytes. We, on the other hand, focus on exploiting redundancy to accelerate the processing time of network-layer applications.

Network-layer applications broadly falls under two categories: header processing applications and payload processing applications[11]. Header processing applications use only packet headers for their processing. A common example of header processing application is packet forwarding (IP-Lookups). In packet forwarding, packets are forwarded by a router using the destination IP address of the packet. Payload processing applications, on the other hand, inspect bytes from the packet payload. Intrusion Detection Systems (IDS) and Virtual Private Networks (VPN) are common examples of payload processing applications. An IDS detects attacks by inspecting packet payload for known attack strings. So pattern matching of attack strings is performed on the payload bytes. This is performance critical since the payload size can be huge, and also due to the huge database of attack strings. So earlier works[6, 7, 12, 14, 16, 28, 29, 33, 36, 37, 38, 39] have investigated mechanisms to accelerate the pattern matching of attack strings in an IDS.

The possibility of significant redundancy in the payload along with the criticality of pattern matching of payload bytes has motivated this work. The broad approach we have taken is to dynamically identify redundant bytes in the payload and skip their redundant processing. While this work focusses on accelerating IDS processing, the approach is extensible to other payload processing applications. So our work on IDS is a demonstration to exploit redundancy in payload processing.

Snort uses the Aho-Corasick algorithm[2] to detect attack strings. This algorithm first converts attack strings into a Finite State Machine (FSM), and then the FSM is traversed with the payload bytes. So the FSM traversal of payload bytes forms the core of IDS, and we explore mechanisms to accelerate this FSM traversal. In order to skip redundant processing, the redundant bytes need to be identified. So we propose a mechanism using a look-up table that identifies the redundant chunk of payload bytes. Once the redundant chunk is identified, we skip past their redundant FSM traversal. Further, we accelerate redundancy identification by performing it concurrently with other Snort functionalities.

We have implemented our proposed mechanisms in the Snort March-2011 release. We have also evaluated it on

an Intel Core i3 with four cores. Our performance results indicate that all traces without exception contain significant redundancy. The table look-up overhead, however, cancels out the gain obtained on eliminating the redundant processing. But the overhead also gets amortized on increasing the redundant chunk length (RL). An increase in RL means lesser redundancy, and hence it is a performance redundancy trade-off. We observe performance gain when the look-up overhead is minimal and for large RLs. This motivates us to investigate a dynamic heuristic that dynamically adapts the RL depending on the performance. The proposed dynamic heuristic provides up-to 16% performance improvement over the FSM traversal used in Snort.

The rest of the paper is organized as follows. Section II provides a brief background on IDS and discusses the Aho-Corasick algorithm. We describe our proposed mechanism to identify and exploit redundancy in Section III. The evaluation methodology used in this work is explained in Section IV. Section V discusses the results. Section VI provides the literature survey in this area. Finally, Section VII concludes this work.

II. Intrusion Detection Systems

Intrusion Detection Systems (IDS) are used to secure the systems in a network. By monitoring the traffic in real time, an IDS detects and also takes preventive action against suspicious activities. There are broadly two types of IDS: anomaly-detection IDS and misuse-detection IDS. An anomaly based IDS functions by detecting anomalous deviation in system parameters. An example of an anomaly is on encountering abnormal series of system calls[17]. So the anomaly detection system uses heuristics to detect this weird program flow, and subsequently takes the needed action. Chandola et al. [8] provides an in-depth survey on anomaly-based IDS. While anomaly based IDS are important and very relevant, in this work we focus on misuse-detection IDS.

Misuse-detection IDS detect malicious activities by inspecting packet payloads for attack strings. These attack strings can also be viewed as signatures, and are byte patterns that have regularly occurred in earlier attacks. As an example, the payload in a cross site scripting attack (XSS) [25] on Microsoft Windows 7 commonly contains the byte pattern '*3A 2F 2F*'. So the Snort IDS searches for this byte pattern in the packet payload. This is equivalent to performing pattern matching of attack strings on the payload bytes. It is also very computationally very expensive. Table I shows the time spent by Snort in the pattern matching module. This is collected using GNU gprof utility [27]. We see that the pattern matching module dominates the execution time of Snort. So we concentrate on accelerating the pattern matching in Snort.

Snort uses the Aho-Corasick algorithm[2] to detect attack strings in the packet payload. This algorithm first constructs a FSM from the attack strings and later traverses the FSM with payload bytes. The main advantage of the Aho-Corasick algorithm is that it guarantees linear-time search irrespective of the number of strings. We provide a brief overview of the Aho-Corasick algorithm with an example. Figure 1 shows the Aho-Corasick FSM built using strings: **ha, he, she, his, him shed**. The FSM is built in two stages, in the first stage characters from the strings are added to the FSM. In the second stage failure transitions are added. Failure transition occurs when the suffix of a string matches the prefix of another string. For example, the transition from state 5 to state 4 is a failure transition. We show failure transitions with dotted lines. For figure clarity, we only show failure transitions from second-level (states 3, 4, 5, 9) onwards. Additionally note that the states 3, 6, 7, 8, 9 , and 10 also correspond to string matches of their respective strings. Once this FSM is built, then it is traversed with bytes from the payload.

Data-sets	% Execution Time
Week-1	64.64
Week-2	65.28
Week-3	65.11
Local Honeypot	61.44

TABLE I: Time Spent in the Pattern Matching Module by Snort.

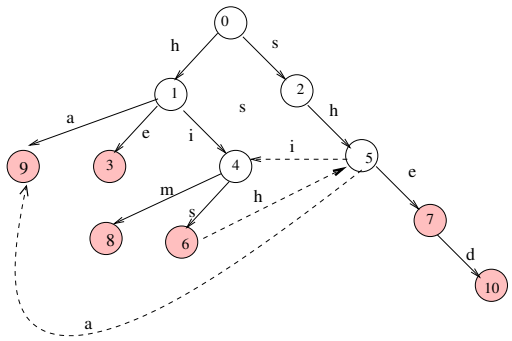


Fig. 1: Example of the Aho-Corasick FSM.

While linear-time search is a powerful advantage, however there are performance/memory issues associated with the algorithm. The FSM thus built gets very bloated. For instance the unoptimized FSM built using the March-2011 release requires 630 MB of memory. So Snort uses various optimizations to reduce this huge memory footprint, and compresses the FSM using a sparse array storage. This and other optimizations(for more details [22]) significantly reduces the memory footprint to 35 MB. However, the execution time for traversing the FSM is also increased. Another issue with the Aho-Corasick algorithm is the sequential nature of the FSM traversal. The FSM traversal of an input byte is dependent on its previous bytes. So it significantly narrows down any possibility of a parallel implementation of the Aho-Corasick algorithm.

Attack strings in IDS are also written as regular expressions, unlike the fixed strings in the Aho-Corasick algorithm. In order to traverse these regular expressions, they need to be converted to Finite Automatas (NFA or DFA). It should be noted that the Finite Automatas are similar to the Aho-Corasick FSM. The Aho-Corasick FSM can even be viewed as a DFA. We have concentrated on the Aho-Corasick algorithm since it dominates

the execution time in the evaluated traces. However, it is important to stress that the mechanisms developed in this work are directly applicable to regular expressions.

The execution flow of a packet in Snort consists of many stages and pattern matching is the final stage. In the prior stages to pattern matching, among the many steps Snort performs also includes packet re-assembly [13]. Re-assembly is needed as attack strings may be cleverly spread across packets by an adversary to evade an IDS. So an IDS commonly reassembles packets, and then performs the pattern matching on the reassembled chunk. We term this reassembled chunk of payload used by the pattern matching module as a datagram.

III. Our Contribution

We aim to eliminate the redundant processing of datagram bytes. In particular we eliminate redundant traversals of the FSM due to redundant bytes. We illustrate this more clearly with an example. Consider the following bytes: **s h i a b c s h i i** as input to the FSM in Figure 1. Datagram bytes traverse the FSM from an **input state** to its **output state**. Figure 2 shows the input and output state corresponding to these bytes.

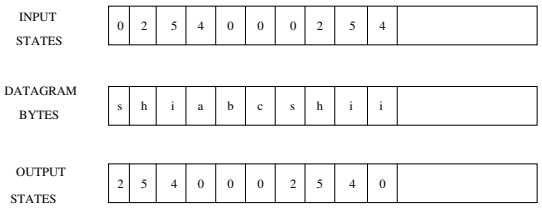


Fig. 2: FSM Traversal with Datagram Bytes.

We observe in the figure that **s h i** are redundant bytes and so they lead to redundant FSM traversal. Our goal is to eliminate this redundant processing. In order to do so, we first need to identify the redundant bytes in the packet. Once the redundant bytes are identified, then their FSM traversal is skipped. So our proposal consists of dynamically identifying redundant bytes and accelerating their processing. We first present a mechanism to dynamically identify redundant bytes.

A. Redundancy Identification

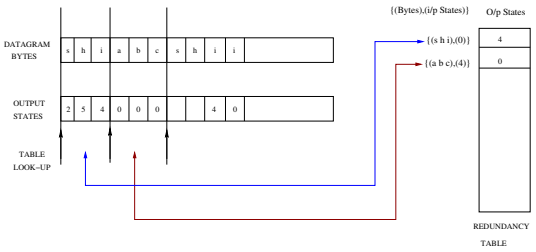


Fig. 3: Redundancy Identification

We skip the redundant FSM traversal using the redundancy table. The redundancy table is indexed using a chunk of datagram bytes. This chunk of bytes forms the unit of redundancy, and we term **Redundancy Length** (RL) as the length of this chunk. RL is a key design factor and significantly affects the performance. A higher RL results in larger strides when we traverse the FSM, and this translates into performance gains. On the contrary, it also capture fewer redundant bytes.

Redundant bytes alone are not sufficient for eliminating redundant processing, the input state is also important. If the input states are different, then there is no guarantee in FSM traversal correctness on skipping the intermediate states. So the table is indexed using a combination of redundant bytes and the input state. Table entries store the final state of redundant bytes and match states if any.

Figure 3 shows an example of the redundancy-aware FSM traversal. In this example, the FSM built in the earlier section (refer to Figure 1) is traversed with datagram bytes. At the first datagram byte, we look-up the table using the datagram bytes (s h i) and the initial state (0). Since there is no matching entry, we continue with the regular FSM traversal of the next byte. The redundancy table is also updated for this entry, $\{$(s h i), (0)$\}$, with the output state. Since the FSM traversal of **s h i**, leads to the output state **4**. We store this output state corresponding to the entry, $\{$(s h i), (0)$\}$, in the table. Note that when the datagram bytes, **s h i**, is encountered the second time, the table look-up is successful. The output state (4) is retrieved from the table, and the intermediate bytes are skipped. Thus the FSM is traversed with the final byte, **i**, and with the input state set to **4**. This way we eliminate the redundant processing due to the redundant FSM traversal.

The redundancy table is implemented in software using standard libraries. In our evaluation we notice that table operations (look-ups and updates) significantly impacts the performance. Hence, table operations i.e., look-ups and updates, are performed not for every byte but at regular intervals. This can also be viewed as systematic sampling of datagram bytes. In this example, we perform sampling with a sampling interval set to 3 (RL).

B. Accelerating Processing of Redundant Bytes

In our redundancy-aware FSM traversal, table look-ups and updates are performed in tandem with the regular FSM traversal. So in Figure 3, there are three table look-ups and updates performed with the FSM traversal. These are overheads that add to the execution time of the regular FSM traversal. So we investigate mechanisms to minimize these overheads. If we examine table operations a bit more closer, we observe the following. The aim of table updates is to identify and capture redundancy, while table look-ups exploits redundancy. So only table look-ups are needed with the regular FSM traversal. We can delay table updates after the FSM traversal.

Further, notice that the redundancy table update operation is completely independent of any IDS processing. So table updates can be performed simultaneously with other IDS functionalities. This can also be viewed as two parallel threads. The Snort thread which is also the main thread, performs the regular IDS processing, while the Redundancy thread identifies and captures the redundancy.

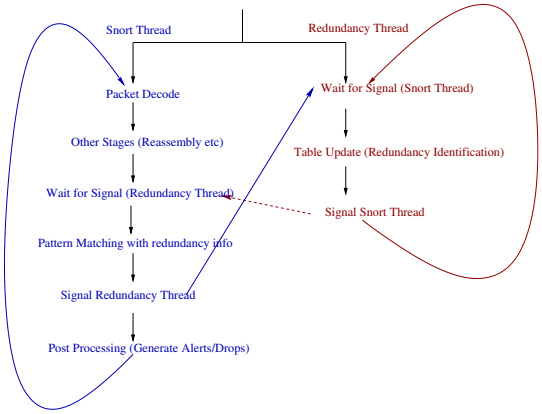


Fig. 4: Thread Functionalities and Interactions

We explain briefly the functionalities and interactions of these threads with an example. Figure 4 shows the execution of Snort and Redundancy threads. When a packet arrives in the system, Snort decodes and reassembles the packet. These are standard functionalities performed by Snort in order to improve its effectiveness. Subsequently, the pattern matching module is called where the FSM is traversed using the datagram bytes. Note that the redundancy-aware FSM traversal is performed, with table look-ups at regular intervals. Once the pattern matching is complete for the datagram, a signal is sent to the Redundancy thread. On receiving this signal, the Redundancy thread starts updating the redundancy table.

The Snort thread meanwhile continues with its regular functionalities. This is the post pattern matching phase where action needs to be taken depending on the pattern matching outcome. Actions are site specific and can include alerting the system administrator or dropping packets. This completes the processing of a packet by Snort, and it moves to the next packet. In the meantime, the Redundancy thread may still be updating the redundancy table using the, now, previous packet. So note that a packet lag in processing occurs between the two threads.

The two threads need thread synchronization as there are data structures shared between them. The redundancy table is the obvious example used by both the threads. The Snort thread uses it for FSM traversal, while the Redundancy thread uses it for table updates. These table operations need to be atomic and hence thread synchronization is needed. So the Snort thread only enters the pattern matching module, only when the Redundancy thread signals it. The redundancy thread only signals the Snort thread after completing the table update. Similarly, the reverse also holds w.r.t signaling by the Snort thread.

The Redundancy thread also requires the datagram bytes buffer and output states for updating the redundancy table. The output state array is generated along with the FSM traversal by the Snort thread. Subsequently, this array is used by the Redundancy thread. The datagram buffer cannot be shared in a similar manner, as it can result in an interesting violation due to the packet lag. Consider the scenario when the Snort thread is reassembling a packet. So it overwrites the datagram buffer which contained the *reassembled packet*. At the same time, the Redundancy thread may be updating the redundancy table with the now *previously reassembled packet*. So it results in a sharing violation of the datagram buffer. In order to avoid this scenario, we make the datagram buffer private to each thread. Hence a copy of the datagram buffer is created by the Snort thread just before it signals the Redundancy thread.

We have implemented the redundancy table in software using Unordered Maps of the Boost Library [18]. Boost Unordered Maps have $\mathcal{O}(1)$ complexity for table look-ups. It is in contrast to C++ Standard STL map which has $\mathcal{O}(\log n)$ complexity. This further motivates us to use Boost Unordered maps.

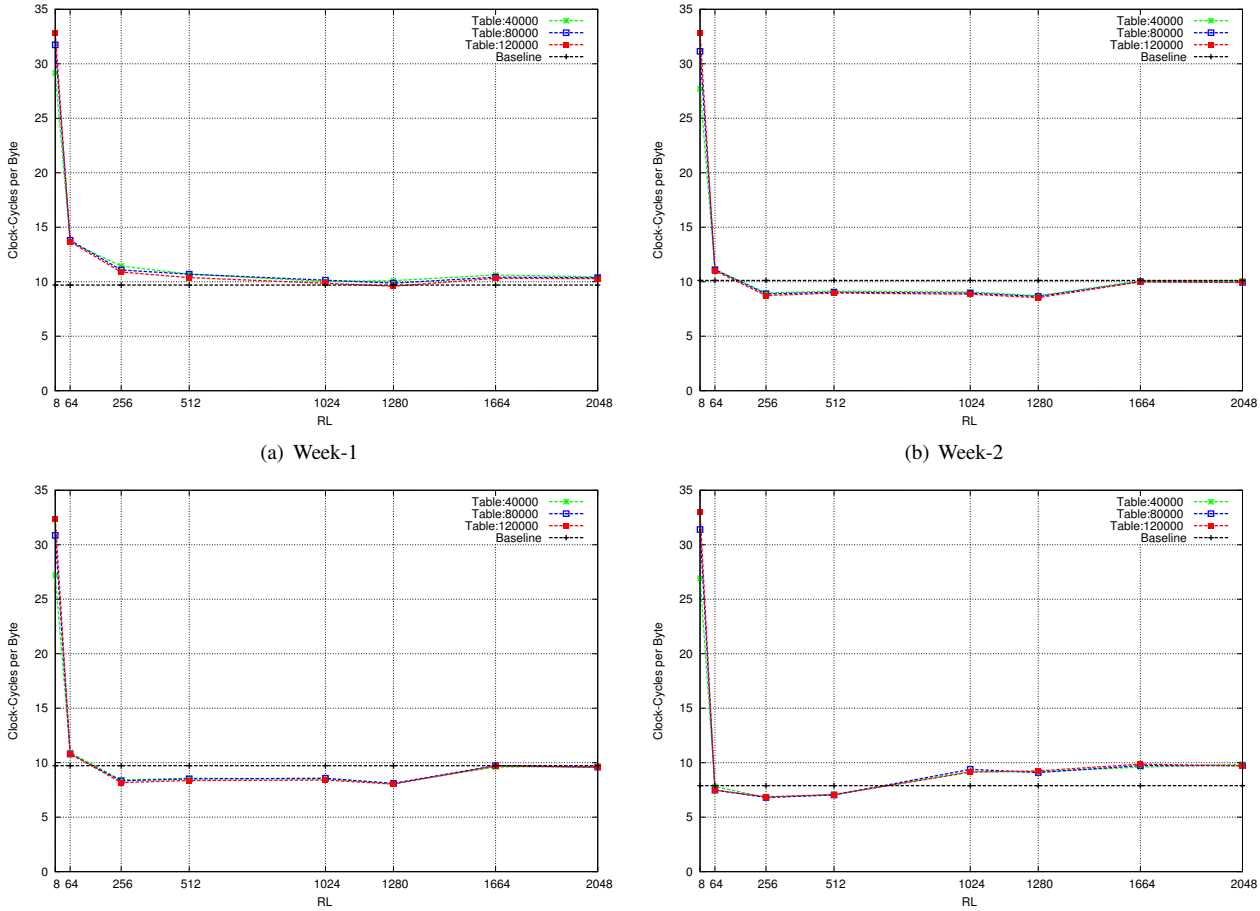
IV. Evaluation Methodology

We have used the metric, Percentage of FSM Traversals Skipped, to measure the redundant processing. It is the number of redundant bytes skipped in the FSM traversal, and is normalized to the total number of datagram bytes in the trace. For instance, the Percentage of FSM traversals skipped in Figure 3 is (2/10)*100=20%. The FSM traversal of **s h** are skipped due to the redundant **s h i** bytes.

We have also measured the performance as the time taken for the FSM traversal by the pattern matching module. So the POSIX clock_gettime() [26] has been used. This clock has a resolution of 1 nano-second. We report the execution time on a per byte basis. This is obtained by dividing the total time taken by the total number of datagram bytes in the trace. We have compared our redundancy-aware FSM traversal with that implemented in the Snort Version 2.9.0.5, March 2011 release. For the evaluation, we have used a Fujitsu Notebook running Ubuntu 11.04 (Linux kernel version 2.6.38-13). This system is an Intel Core i3 with 4 cores and 4 GB RAM. Additionally, all configurations are run three times and we report the average of the three runs.

A. Data-sets

We have used 4 traces in our evaluation. We have deployed a low-interaction Honeypot in collaboration with the Leurrecom project[30]. This Honeypot is in the De-Militarized-Zone (DMZ) of the university LAN. An issue with low-interaction honeypots is that there is hardly any interaction, and we have first-hand experience with Honeyd[31]. However, our experience with the Leurrecom Honeypot indicates that it regular interacts with the outside world. In the Honeypot logs, we observe that there has been interaction with the outside world for at-least 61 days (out of the 3 months it has been deployed). We use this *Local Honeypot* trace in our evaluation. The IDS evaluation traces are from the 1999 MIT Lincoln labs[23] traces. We have used 3 weeks, namely, Week-1, Week-2, and Week-3 traces. Though these are provided by MIT as daily traces for 5 days, we have aggregated the daily traces into a week. We refer to these traces by their respective week. Table II summarizes the traces used in the evaluation.



(a) Week-1

(b) Week-2

(c) Week-3

(d) Local Honeypot

Fig. 5: Execution Time Comparison for Various Traces

Data-sets	Avg Datagram Size(B)	Num Datagrams(M)
Local Honeypot	197	0.69
Week-1	421	2.04
Week-2	433	12.31
Week-3	479	22.96

TABLE II: Traces Used.

For each trace, the average datagram size inspected by the Snort pattern matching module is also reported.

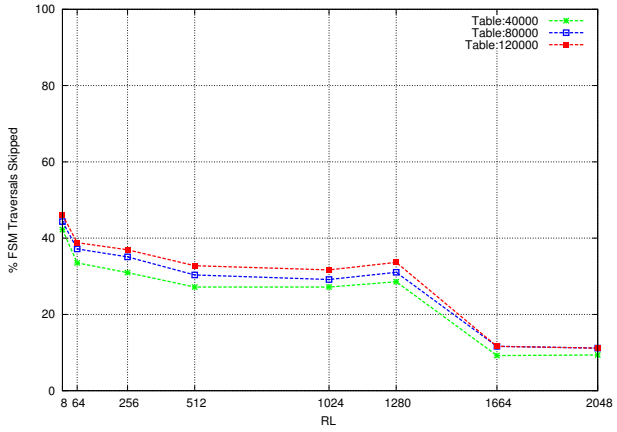
V. Results

Figure 5 shows the execution time of the redundancy-aware mechanism for different RLs and table sizes. The table size has been varied from 40K entries to 120K entries. In this Figure we also compare the performance of our proposal with the base Snort implementation (referred to as Baseline).

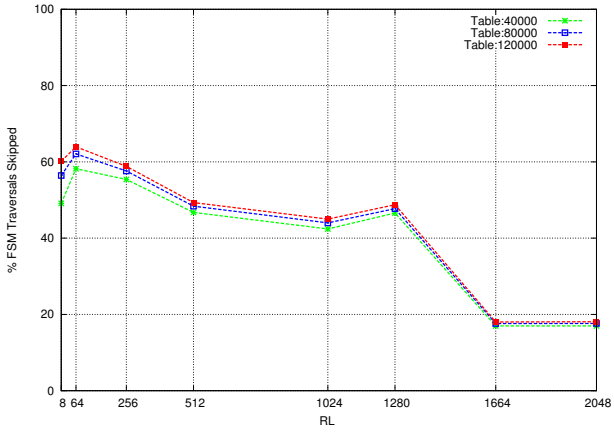
The execution time results for all traces shows an interesting trend, namely, that an increase in RL speeds up the pattern matching. For example in the Week-1 trace, $RL = 8$ requires 32.84 cycles per Byte while RL=2048 needs only 10.28 cycles. In all traces, $RL = 8$ is the most clock consuming configuration. It is also interesting to note that $RL = 8$ also captures a very high redundancy (refer to Figure 6). For example in the above considered trace $RL = 8$ skips 52% of datagram bytes in FSM traversal. This very unusual behaviour of a very high redundancy with a very low performance is due to the overhead in the table look-up. Note that in our redundancy-aware FSM traversal, table look-ups are performed in addition to the regular FSM traversal.

We investigate the table look-up operation in more detail. Internally the table look-up in the Boost library is performed in the following manner. Consider the example in Figure 3, when $\{$(s h i),(0)$\}$ is encountered the second time. For a table look-up, first a hash value of $\{$(s h i),(0)$\}$ is computed using the in-built Boost Hash library. Next this hash value is indexed into the table, and since there is a matching entry, it is retrieved. Further, this entry is checked for hash collisions (false positives). To do so, $\{$(s h i),(0)$\}$ are compared with their equivalents in the table entry. Since they match, the table look-up is successful. The steps outlined above are clearly non-trivial in performance. For example, the hash collision checking is a memory comparison operation (memcmp in C String Library).

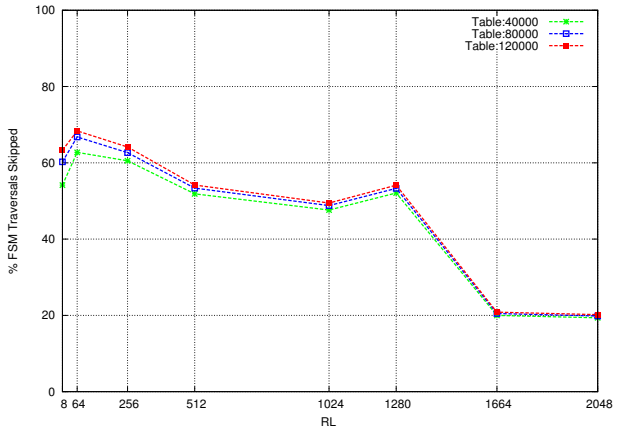
We now study the overhead due to table look-ups. In order



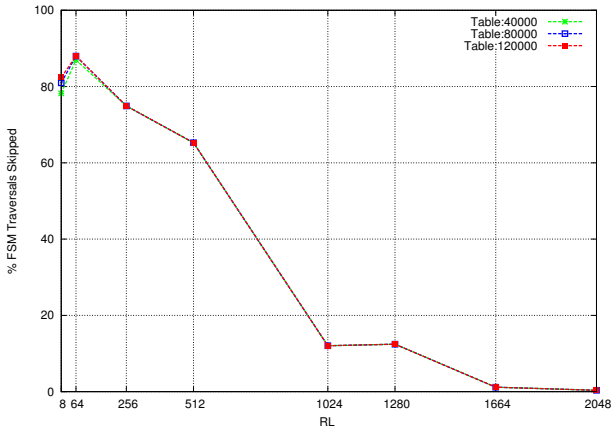
(a) Week-1



(b) Week-2



(c) Week-3



(d) Local Honeypot

Fig. 6: Redundancy Results for Various Traces

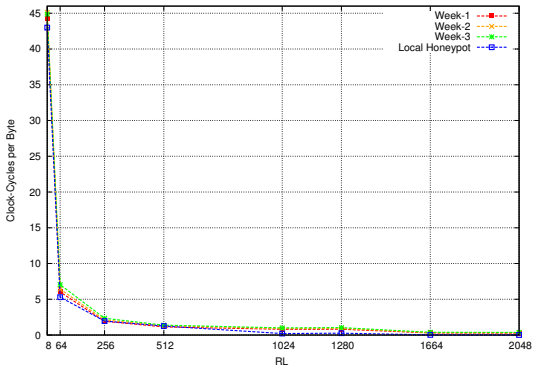


Fig. 7: Table Look-up Overhead

to meaningfully compare the look-up overhead for different RLs, we report it on a per indexed byte basis. We explain it clearly with an example. Consider the datagram in Figure 3 with $RL = 3$ and $RL = 6$ and table look-up time of T_3 and T_6 respectively. We report the look-up overhead incurred for these RLs as $T_3/9$ (due to 3 look-ups) and $T_6/6$ respectively. Figure 7 shows the look-up overhead for different RLs. We clearly see that the table look-up overhead is high for low RL values, and the highest for RL=8. Since $RL = 8$ incurs the maximum overhead, its performance is relatively the worst. However, it reduces on increasing the RL, so the table look-up overheads gets amortized with large RL values.

We have used this indirect method of inferring the table look-up overhead due to the following reason. If a direct break-down of various operations (like table look-up, thread synchronization etc) in the FSM traversal is attempted, then we observe that the clocks thus inserted significantly dominate the performance. So it does not provide a realistic impact of the various operations in the FSM traversal.

The table look-up overhead results clearly indicate that larger RLs incur relatively lesser overhead, and so provide better performance. Additionally on analyzing the redundancy results (refer to Figure 6), we observe that moderate to significant redundancy exists at large RLs. For instance in the Week-2 trace, $RL = 1280$ skips 46% of the datagram bytes in the FSM traversal. We also observe for $256 \leq RL \leq 1280$ that the redundancy-aware FSM traversal outperforms the Baseline (refer to Figure 5). This performance gain is due to the large

fraction of bytes being skipped with relatively lesser table look-up overhead. Note that for $RL > 1280$ very few bytes are skipped of FSM traversal, and so there is no performance gain. The performance of the Week-1 and Week-3 trace also show a similar behaviour as the Week-2 trace. However, note that in case of the Week-1 trace the redundancy at large RLs is not as significant as Week-2 and Week-3 traces. So the performance improvement in the Week-1 trace is not as noticeable as the other traces.

In case of the Local Honeypot trace, $64 \leq RL \leq 512$ outperforms the Baseline. But for $RL > 512$, there is a drop in performance due to fewer table look-ups. Note that for relatively small datagrams, large RLs results in fewer table look-ups[1]. So for $RL = 512$, we observe that 68% of datagram bytes are looked-up. However for $RL = 1024$ only 14% of datagram bytes are looked-up. So large RL values do not provide any performance benefit for the Local Honeypot trace.

In our results we have also varied the table size from 40K entries to 120K entries. As can be noticed from the results, there is no significant gain in using larger sized redundancy table.

Our results can be summarized as follows. The redundancy-aware mechanism provides performance benefits when moderate to significant redundancy is present in a trace at large RL values. Furthermore, performance improvement is obtained when the overhead in the table look-up is not high. So the trace characteristic governs the redundancy and hence the performance. Hence in-order to dynamically determine the RL, we explore a dynamic heuristic that adapts the RL value at run-time.

A. Dynamic Heuristic

Our earlier results indicate that the best performing RL is dependent on the trace characteristic. For instance, $256 \leq RL \leq 512$ provides performance gains in the Local honeypot trace. On the other hand, $256 \leq RL \leq 1280$ provides performance improvement for the remaining traces. So the dynamic heuristic must be able to dynamically identify these RL ranges and set the RL value accordingly.

Algorithm 1 outlines the steps of our proposed dynamic heuristic. The dynamic heuristic analyzes the performance of the FSM traversal at every 10M datagram bytes. It compares the performance of the current RL($Perf_{Cur\_RL}$) with the previously set RL($Perf_{Prev\_RL}$). The performance is measured as the average number of clock-cycles for the FSM traversal of a datagram byte. If the dynamic heuristic observes a performance drop then the RL is accordingly modified. So if at the last epoch, RL was increased then RL needs to be decreased and vice-versa. The RL value is also set in a similar manner on a performance gain.

In our evaluation, we have varied the stride for increasing (INC_STRIDE) and decreasing (DEC_STRIDE) the RL. Our study shows that $INC\_STRIDE = 256$ and $DEC\_STRIDE = 128$ provides the best performance. We have also explored the interval for invoking the dynamic algorithm. We observe that polling at every 10M datagram bytes provides the best performance. The dynamic heuristic is executed by the Redundancy thread, and the table is also cleared if the RL value is modified.

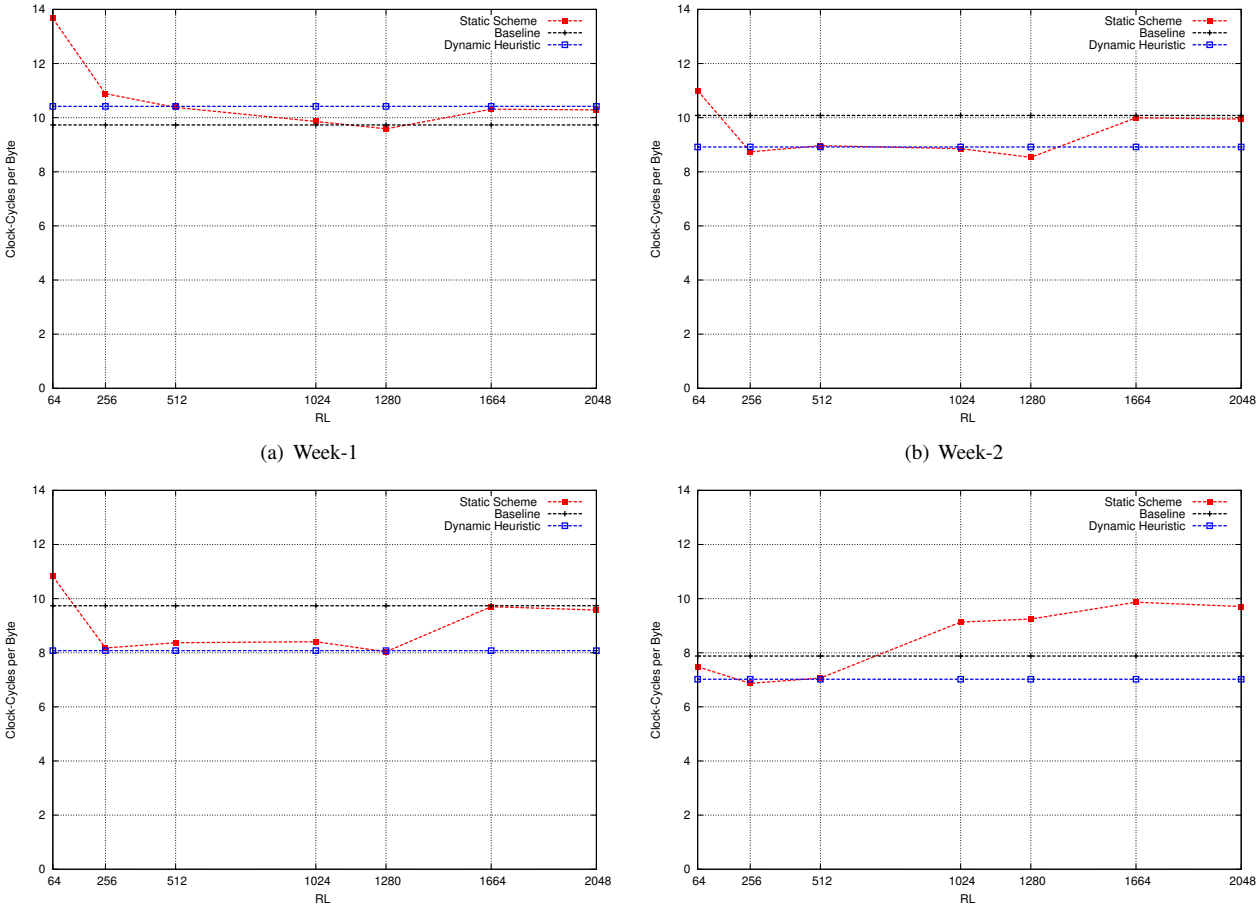
Algorithm 1 Dynamic Algorithm to Set the RL.

```
if Total_Num_Datagram_Bytes mod 10^7 then
  if Perf_Cur_RL − Perf_Prev_RL < 1 then
    if RL_Increased then
      RL = RL - DEC_STRIDE {Perf↑ when RL↓}
      RL_Increased = 0
    else
      RL = RL + INC_STRIDE {Perf↑ when RL↑}
      RL_Increased = 1
    end if
  end if
else
  if Perf_Prev_RL − Perf_Cur_RL < 1 then
    if RL_Increased then
      RL = RL + INC_STRIDE {Perf↑ when RL↑}
      RL_Increased = 1
    else
      RL = RL - DEC_STRIDE {Perf↑ when RL↓}
      RL_Increased = 0
    end if
  end if
  Perf_Cur_RL = Perf_Prev_RL
end if
```

Figure 8 shows the performance of our dynamic heuristic with respect to the Baseline and the static scheme. Note that the static scheme uses a fixed RL value and is identical to the redundancy-aware mechanism previously discussed. Our results shows that our proposed dynamic heuristic is able to dynamically adapt to the trace. So in case of Week-2 and Week-3 traces, the dynamic heuristic provides 11% and 16% performance improvement over the baseline respectively. The Local Honeypot trace provides 13% performance improvement over the baseline. The performance gains thus obtained by the dynamic heuristic is due to the detection of the optimal RL ranges for the traces. Thus for the Local Honeypot trace, the RL value lies between 256 and 512, and is only changed less than 3 times in the entire run. A similar behaviour is also observed for the Week traces, with the RL varying between 256 and 1280. However, the Week-1 trace shows a mild performance degradation of 5%. This degradation is due to the relatively lesser redundancy present in the Week-1 trace.

As mentioned earlier, the redundancy table is implemented using the Unordered map in the Boost library. The redundancy table can also be implemented as a specialized logic for accelerating the IDS processing. Earlier works have explored dedicated hardware logic for accelerating the FSM traversal in an IDS[7, 10, 12, 14, 28, 29, 33, 36, 37, 39]. Thus the table can be viewed as a dedicated redundancy cache, and the table look-

[1] $Num_{look-ups} = \lceil (Datagram\_Size - RL + 1)/RL \rceil$



(a) Week-1

(b) Week-2

(c) Week-3

(d) Local Honeypot

Fig. 8: Execution Time Comparison for the Dynamic Heuristic

up operations can also be viewed in terms of cache accesses. The hash computation in our table look-up implementation is equivalent to indexing the sets of a cache. Further, the memory comparison operation in our table look-up implementation is a tag comparison operation in a cache. Since a specialized hardware structure will not incur the software overheads as discussed earlier. So deploying such a specialized structure will help to even further improve the performance.

VI. Related Work

The literature related to this work falls broadly under temporal locality studies in network traffic or optimizations in an IDS. We briefly summarize the prior work done in each of these areas.

Redundancy in payload has been studied recently[1, 4]. Anand at al [4] study the redundancy in payload in a variety of enterprise traces. They further investigate various sampling techniques for detecting and exploiting the redundancy. Their goal is to maximize bandwidth saving while transmitting the payload. Similarly Aggarwal et al.[1] have proposed and evaluated a performance efficient sampling technique for bandwidth saving. In contrast to these works, the aim of our work is to eliminate the redundant processing in payload processing applications. Further, we have operated on datagram bytes (also viewed as a reassembled packet), while [1, 4] study at the granularity of packet payload. Note that the packet size is typically less than 1500 B[2], while reassembled packets can be up-to 64 KB. This difference in data granularity is also reflected in the redundancy results. While we observe moderate to significant redundancy for $RL \geq 256$, their study restricts RL to value less than 128.

Redundancy has also been studied and exploited in packet headers. Notably, packet forwarding (look-ups) exhibits redundancy in destination IP addresses. So [5, 15, 24] have studied memory architectures for efficient packet look-ups. Similarly, other packet header fields also exhibit redundancy, and it is exploited in packet classification[20, 34]. Redundancy in HTTP requests sent to web servers has been extensively studied. Web caches are as a consequence of redundancy in HTTP requests. There have been several interesting studies in this area for at-least the last fifteen years. Notably Almeida et al.[3] comprehensively characterize the redundancy in web

[2]in case of Ethernet

requests.

Earlier works on FSM in IDS have broadly either investigated mechanisms to compactly store the FSM, and/or accelerate its traversal. As reported earlier, the FSM built using the base Aho-Corasick algorithm can get very bloated. So Tuck et al.[38] propose using a bitmap structure for states in the FSM. They further investigate various mechanisms including eliminating failure edges from the FSM. Becchi et al.[6] investigate merging similar states in the FSM. Kumar et al.[16] propose a compact FSM storage that eliminates failure edges. Randy et al.[35] propose various FSM enhancements for eliminating duplicate states in the FSM.

There have also been works on accelerating the FSM traversal. Brodie et al.[7] propose traversing the FSM with multiple bytes. So they build the FSM so that it traverses with 2 input bytes at a time (as opposed to 1 byte in Figure 1). Tan et al.[37] have investigated techniques to concurrently traverse the FSM. So they propose a bit-wise traversal instead of the standard byte-wise traversal. Shenoy et al.[33] observe that the root-node is the most frequently accessed node. So they propose mechanisms that includes a pipelined architecture for accelerating the root-node traversal. Luchaup et al.[19] speculate the FSM traversal. Speculation is used since there are only few unique states traversed by the FSM. Note that there is a difference between our redundancy-aware FSM traversal and using speculation for FSM traversal. Speculation requires a roll back mechanism, while our redundancy-aware FSM traversal does not require any roll back as traversal correctness is guaranteed. Hua et al.[14] investigate using a variable stride FSM and so the FSM is also built accordingly. They tune their traversal specifically for TCAMs. Anirban et al.[21] study clustering the regular expressions based on their popularity. They dynamically build the FSM (DFA) only for frequently used regular expressions.

It is very interesting to note that our redundancy-aware FSM traversal is almost orthogonal to all these techniques. So it can complement almost all these techniques in accelerating the FSM traversal.

VII. Conclusion

In this work, we have investigated the temporal locality in the packet payload. We observe significant locality in all traces analyzed, and study mechanisms to exploit it. IDS is an example of a payload processing application which is critically dependent on processing of payload bytes. So we take IDS as a case study to exploit temporal locality. The string matching module in IDS is the critical module for performance in Snort IDS. This module inspects packet payload for attack strings from a database of attack strings. Snort IDS commonly uses the Aho-Corasick algorithm for string matching. So it builds a FSM from the attack strings and traverses the FSM using the payload bytes.

We propose a redundancy-aware FSM traversal that dynamically identifies redundant payload bytes and skip their redundant processing. We further parallelize our mechanism by performing the redundancy identification concurrently with stages in Snort packet processing. We have implemented our redundancy-aware pattern matching in Snort, and have evaluated on a multi-core in a desktop environment. Our performance results indicate that all traces without exception contain significant redundancy. The look-up overhead, however, cancels out gains thus obtained. But the overhead also get amortized on increasing the RL. An increase in RL also means lesser redundancy and so it is a performance redundancy tradeoff. We observe performance gain when the look-up overhead is minimal and for large RLs ($RL \geq 256$). This motivates us to propose a dynamic heuristic that dynamically modulates the RL depending on the performance. Our results indicate that this dynamic heuristic provides up-to 16% performance improvement over the regular FSM traversal used in Snort.

An extension of this work is to study a redundancy-aware mechanism in the IPSec algorithms used by VPNs. A potential issue, however, could be the possibility of compromising the confidentiality/integrity of data. So the redundancy-aware mechanism will have to take into consideration this potentially sticky issue. Furthermore, the key insight of our study is the presence of significant redundant processing in all traces. This points to the possibility of predicting the processing (header or payload) of packet. Packet processing prediction can be useful during packet bursts, when congestion may force the router to drop packets.

VIII. Acknowledgements

This work has been supported by the following grants: TIN2010-18368, TIN2007-61763, and SGR2009-1250. This work has been supported by the Spanish Ministry and Intel Corporation.

References

[1] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: An End-system Redundancy Elimination Service for Enterprises. *In Proceedings of the 7^{th} USENIX Conference on Networked Systems Design and Implementation*, 2010.

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

[3] V. Almeida, A. Bestavros, M. Crovella, and A. D. Oliveira. Characterizing Reference Locality in the WWW. *In Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, 1996.

[4] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundancy in Network Traffic: Findings and Implications. *In Proceedings of the 11^{th} International Conference on Measurement and Modeling of Computer Systems*, 2009.

[5] J.-L. Baer, D. Low, P. Crowley, and N. Sidhwaney. Memory Hierarchy Design for a Multiprocessor Look-up Engine. *In Proceedings of the 12^{th} Parallel Architecture and Compiler Techniques*, 2003.

[6] M. Becchi and S. Cadambi. Memory Efficient Regular Expression Search Using State Merging. *In Proceedings of the 26^{th} IEEE Infocom*, 2007.

[7] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A Scalable Architecture For High Throughput Regular-Expression Pattern Matching. *In Proceedings of the 33^{rd} Annual International Symposium on Computer Architecture*, 2006.

[8] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Computing Survey*, 2009.

[9] F. Chang, W. Feng, and K. Li. Approximate Caches for Packet Classifications. *In Proceedings of the 23rd IEEE Infocom*, 2004.

[10] C. Clark and D. Schimmel. Scalable Pattern Matching for High Speed Networks. *In Proceedings of the 12th Annual Symposium on Field Programmable Custom Computing Machines*, 2004.

[11] D. Comer and L. Peterson. *Network Systems Design Using Network Processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1 edition, 2003.

[12] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Implementation of a Deep Packet Inspection Circuit using Parallel Bloom Filters in Reconfigurable Hardware. *In Proceedings of the IEEE Hot Interconnects (HOTi)*, 2003.

[13] S. Dharmapurikar and V. Paxson. Robust TCP Stream Reassembly in the Presence of Adversaries. *In Proceedings of the 14th Conference on USENIX Security Symposium*, 2005.

[14] N. Hua, H. Song, and T. V. Lakshman. Variable-Stride Multi-Pattern Matching For Scalable Deep Packet Inspection. *In Proceedings of the 28th IEEE Infocom*, 2009.

[15] A. Kumar, L. De Carli, S. J. Kim, M. de Kruijf, K. Sankaralingam, C. Estan, and S. Jha. Design and Implementation of the PLUG Architecture for Programmable and Efficient Network Lookups. *In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[16] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. *In SIGCOMM Computer Communication Review*, 2006.

[17] W. Lee and S. J. Stolfo. Data Mining Approaches for Intrusion Detection. *In Proceedings of the 7th USENIX Security Symposium*, 1998.

[18] Boost C++ Library and Unordered Maps. http://www.boost.org/doc/libs/1_46_0/doc/html/unordered.html.

[19] D. Luchaup, R. Smith, C. Estan, and S. Jha. Multi-byte Regular Expression Matching with Speculation. *In Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, 2009.

[20] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. *In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2010.

[21] A. Majumder, R. Rastogi, and S. Vanama. Scalable Regular Expression Matching on Data Streams. *In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[22] Snort Online Manual. http://manual.snort.org.

[23] MIT Lincoln Labs, DARPA Intrusion Detection Evaluation. http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/.

[24] J. Mudigonda, H. M. Vin, and R. Yavatkar. Overcoming the Memory Wall in Packet Processing: Hammers or Ladders. *In Proceedings of the 2005 ACM/IEEE Symposium on Architecture for Networking and Communication Systems*, 2005.

[25] Cross Scripting Attack on the MHTML Protocol Handler in Microsoft Windows. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0096.

[26] Clock_Gettime Manual Page. http://linux.die.net/man/3/clock_gettime.

[27] GNU Gprof Manual Page. http://sourceware.org/binutils/docs/gprof/.

[28] P. Piyachon and Y. Luo. Efficient Memory Utilization on Network Processors for Deep Packet Inspection. *In Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, 2006.

[29] P. Piyachon and Y. Luo. Compact State Machines for High Performance Pattern Matching. *In Proceedings of the 44th Annual Design Automation Conference*, 2007.

[30] F. Pouget, M. Dacier, and H. Pham. Leurre.com: On the Advantages of Deploying a Large Scale Distributed Honeypot Platform. *E-Crime and Computer Conference*, 2005.

[31] N. Provos. Honeyd - A Virtual Honeypot Daemon. *In 10th DFN-CERT Workshop, Hamburg, Germany*, 2003.

[32] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. *In Proceedings of the 13th USENIX Conference on System Administration*, 1999.

[33] G. Shenoy, J. Tubella, and A. Gonzalez. A Performance and Area Efficient Architecture for Intrusion Detection Systems. *In Proceedings of the 25th IEEE International Conference on Parallel and Distributed Processing Symposium*, 2011.

[34] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification using Multidimensional Cutting. *In Proceedings of the 2003 SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2003.

[35] R. Smith, C. Estan, S. Jha, and I. Siahaan. Fast Signature Matching Using Extended Finite Automaton (XFA). *In Proceedings of the 29th IEEE Symposium on Security and Privacy*, 2008.

[36] T. Song, W. Zhang, D. Wang, and Y. Xue. A Memory Efficient Multiple Pattern Matching Architecture for Network Security. *In Proceedings of the 27th IEEE Infocom*, 2008.

[37] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. *In Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[38] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. *In Proceedings of the 23rd IEEE Infocom*, 2004.

[39] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. *In Proceedings of the 12th IEEE International Conference on Network Protocols*, 2004.