A Performance and Area Efficient Architecture for Intrusion Detection Systems

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*—Intrusion Detection Systems (IDS) have emerged as one of the most promising ways to secure systems in network. An IDS operates by scanning packet-data for known signatures and accordingly takes requisite action. However, scanning bytes in the packet payload and checking for more than 20,000 signatures becomes a computationally intensive task. Additionally, with signatures doubling almost every 30 months, this complexity will aggravate further.**

IDS commonly uses the Aho-Corasick state machine based search to scan packets for signatures. However, the huge size of the state machine negatively impacts the performance and area efficiency of the underlying hardware. In this work, we propose novel mechanisms to compactly store the state machine thereby improving the area efficiency. We observe over 2X reduction in area for storing the state machine in comparison to BS-FSM [19]. We investigate various approaches to improve the performance efficiency. We pipeline the processing of consecutive bytes accessing the upper-most level, the frequently accessed level, of the state machine. In order to further enhance the performance efficiency, we use a dedicated hardware unit specifically tuned for traversal using our proposed storage mechanism. We observe that our proposed architecture outperforms BS-FSM based approaches [13, 14, 19].

I. Introduction

Computing systems today operate in an environment of seamless connectivity. The explosion of the Internet has resulted in electronic commerce increasingly dominating business transactions. The interaction between customers and suppliers through Internet has necessitated a strict demand for trust in Internet for successful and secure interaction. Intrusion Detection Systems (IDS) are emerging as one of the most promising ways of providing protection to systems on the network against suspicious activities. By monitoring the traffic in real time, an IDS can detect and also take preventive actions against suspicious activities.

In order to identify malicious traffic, an IDS uses signatures of existing attacks which are applied to packet content flowing through the network. Further, these signatures can be located anywhere in the packet, and hence the packet payload needs to be scanned. Additionally, there are more than 23,000 strings in Snort [15]. This puts a tremendous computational requirement on the IDS, both in terms of the amount of time to process a packet and the amount of memory needed to store the signatures. Signatures can be viewed as *strings* that need to be matched. So, in order to perform multiple string matching, Snort uses the Aho-Corasick algorithm [1]. This algorithm works by constructing a state machine based on a set of strings that need to be matched. Once this state machine is constructed, incoming bytes from the packets are used to traverse through this state machine. However, there are area and performance issues in realizing a practical implementation.

In order to address these concerns, we present a novel architecture for IDS. We propose a novel way of storing the state machine. This storage consists of a hybrid data-structure with one type of storage for the root-node and another for other-level nodes. We also investigate mechanisms to further reduce the storage size of other-level nodes. However, there are performance and storage size issues with this structure. We investigate various mechanisms to address these concerns. Our novel storage of the state machine results in over 2X reduction in area in comparison to BS-FSM [19]. We investigate various mechanisms to improve the performance efficiency of this architecture. We observe that multiple consecutive bytes access the root-node very frequently. So we propose a dedicated pipelined processing unit to process these consecutive root-node accesses. We further enhance our architecture with the addition of a hardware unit specially tailored to accelerate the traversal of non root-nodes. Our performance results indicate that our proposed architecture outperforms BS-FSM based approaches [13, 14, 19].

The rest of the paper is organized as follows. Section II provides background on Aho-Corasick Algorithm. The related work in this area is discussed in Section III. We present our mechanisms to improve the area efficiency in Section IV. In Section V we present various mechanisms to improve the performance efficiency. The simulation methodology used in obtaining the results is discussed in Section VI. Section VII presents the performance results. Section VIII concludes this work.

II. Background

Snort uses the Aho-Corasick algorithm for string matching [1]. This algorithm works by constructing a state machine based on the set of strings that need to be matched. Once this state machine is constructed, incoming bytes from packets are used to traverse through this state machine. The main advantage using this algorithm, in contrast to other string matching algorithms, is that it guarantees linear-time search irrespective of number of strings. Due to this reason, Snort

and other IDSs commonly use this state machine based search for string matching. We provide a brief overview of the Aho-Corasick algorithm with an example.

Consider the set of strings: **ha, he, she, his, him shed.** Figure 1 shows the corresponding Aho-Corasick state machine constructed from these strings. The state machine is built in

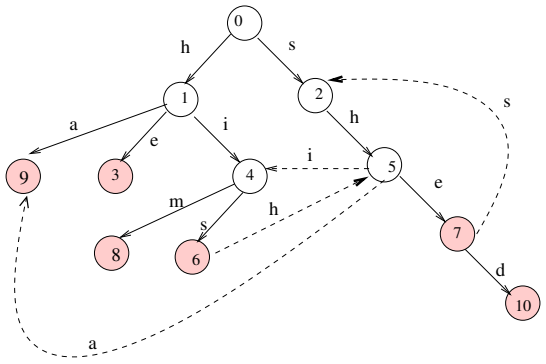


Fig. 1. Example of the Aho-Corasick State Machine.

two stages. In the first stage, characters from strings are added to the state machine. This is done in a way that strings that share a common prefix also share the same set of parents in the state machine. The edges corresponding to this stage are shown as thick lines. Also note that **nodes 3, 6, 7, 8, 9, 10** indicate a match for strings **he, his, she, him, ha, shed** respectively. These nodes also store a pointer to a list of matched strings. For example, **node 7** stores a pointer to the list of its matched strings namely **he, she**.

The second stage in building the state machine consists of inserting failure edges. When a string match is not found, it is possible for the suffix of one string to match the prefix of another, so failure edges need to be inserted. Failure edges are shown with dotted lines. For figure clarity, only a few failure edges are shown. Once this state machine is built, the algorithm traverses it with bytes from packet. In case the byte does not correspond to any of the examined edges, then the traversal is restarted from the root-node.

A few terminology clarifications. Henceforth, the number of outgoing edges from a node is referred to as fan-out. For example, fan-out of **node 5** is 3. Also, any reference to database or string database refers to the Snort string database.

The main advantage of this algorithm is that it runs in linear time to the input string, regardless of number of strings. However, the problem with this algorithm lies in devising a practical implementation. This is again due to the large fan-out of each and every node. Implementing this requires a great deal of next pointers, 256 for each and every node to be exact. This consequently increases the size of the state machine. For example, the state machine built using the September-2007 Snort string database contains more than 42,000 nodes and requires 44 MB of storage space. Additionally, with new attacks being created all the time, the database needs to be regularly updated. This, in turn, results in a growing string database. So, the storage space requirements will keep growing. So one of the implementation issues with Aho-Corasick algorithm is the growing memory area required to store the growing state machine. Another implementation issue is the sequential nature of traversal. The determination of the next state is strictly dependent on the current state. So, multiple bytes from a packet can only be processed in a strict sequential order.

While fixed pattern matching is important so too is regular expression matching. Regular expressions are increasingly used in IDS due to their rich expressive power. We observe that 1751 rules (out of 3816) in the Snort Sept-07 database use regular expression matching. Interestingly in regular expression matching, we observe that Snort first does a fixed string pattern matching on the regular expression prefix. If there is a prefix match only then is the regular expression matcher invoked.

III. Related Work

Broadly speaking, earlier works in this direction can be classified either as hardware or software approaches. Software approaches optimize the data-structure, thereby reducing the size of the state machine. While, hardware approaches accelerate string matching with specialized structures.

Tuck et al [21] study different optimizations to reduce the size of each node in the state machine. They use a 256 bit bitmap which is used in place of 256 next-node pointers. A bit is set in the bitmap if the corresponding character has a valid next-node. They also use path compression to compress the bitmap structure. While this work reduces the memory size requirements, a disadvantage is the additional computational complexity due to compression.

Tan et al [19] reduce the high fan-out by maintaining a bit-level state machine for every bit in the byte. These independent state machines are traversed concurrently, and the output are the bits set in a bit-vector corresponding to matched patterns. Further, an intersection of bit-vectors gives the matched patterns. The benefit of this approach is that it reduces the state machine size, and also provides parallelism since these state machines can be traversed concurrently. The use of parallelism together with reduction in state machine size significantly improves the performance and area efficiency.

Piyachon et al [12] exploit parallelism available in network processors. For example, the Intel IXP-2800 network processor has 16 RISC cores, with each core executing 8 threads. They partition the Snort database among these cores. This approach provides an efficient way of utilizing on-chip memory in a network processor. However, with the string database growing non-linearly, and limited on-chip memory available in network processor, this approach may not be scalable.

Piyachon et al [13] observe that a large percentage (>59%) of states do not have any matched pattern. They further observe that in the case of [19], bit-vector dominates storage space. So they propose heuristics to store bit-vectors only for states with

matching pattern. Additionally, they also decouple storage of the state machine from the bit-vector.

Piyachon et al [14] extend [19] by using a translation table and a CAM instead of bit-vectors. They propose a relabeling algorithm that assigns identical state labels to various states that match the same pattern. Note that these states are on different bit slices of the bit-split FSM [19]. A translation table is used to obtain the matched pattern, and it stores the matched patterns corresponding to the common state label. When state with different labels match a pattern, then a CAM is used. Both the CAM and the translation table are indexed using state labels.

Lin et al [10] observe that there are numerous equivalent nodes in the state machine. Two nodes are defined as equivalent if they have identical input, failure edges and outputs. They propose merging these equivalent nodes by adding a bit-vector to the base structure. Our proposal can be combined with this approach and thus can potentially yield even further reduction in memory area.

Sourdis et al [17] propose pre-filtering for string matching. They observe that it is very rare for a single incoming packet to fully or partially match more than a few tens of strings. Based on this observation, they select a small portion from each string to be used in the pre-filtering step. The result of pre-filtering step is a reduced set of strings that are candidates for full match. Given this reduced set, the second stage is an entire packet matching using the reduced set of strings. Pre-filtering improves throughput of IDS at no additional cost.

Some earlier works also use specialized hardware structures that accelerates string matching. Dharmapurikar et al [6] use bloom filters and Yu et al [22] use TCAMs for string matching. Earlier works have also augmented FPGAs for string matching [4] or exploited available features in commercial FPGAs [16]. The primary advantage in using FPGAs is reconfigurability which is useful when updating the database every few months.

Additionally, there have been numerous works ([2, 9, 18] to cite a few) in reducing the memory space for storing DFAs. Note that DFAs have very similar behaviour to that of the Aho-Corasick state machine. Some of these works also investigate techniques to efficiently traverse the resulting compact DFA.

We first discuss our proposed techniques to improve the area efficiency and later deal with techniques to improve the performance.

IV. Improving Area Efficiency

The bloated size of the state machine (44 MB) is due to the large size of each node (for storing 256 next-node pointers). We observe that the fan-out of nodes in the state machine varies widely. For example, the root-node has a fan-out of 103 while nodes near the leafs have very less fan-out. So for nodes with low fan-out, the storage space is wasted by storing 256 pointers. Additionally, we also observe that the root-node is accessed by up-to 93% of incoming bytes. Armed with these observations, we propose a novel hybrid storage with one type of storage for the root-node and a different storage for other-level nodes. We first explain the other-level node storage.

In our proposed storage, we store a node as the set of its outgoing edges. Each outgoing edge has the following: the corresponding byte, the fan-out of next-node, the offset to the next-node, and rule offset of next-node. The size of each edge using this storage is: 1 B for byte, 1 B for fan-out, 3 B each for next-node and rule offsets, thus making a total of 8 B per edge. A collection of all these outgoing edges forms a node. We illustrate this more clearly with an example.

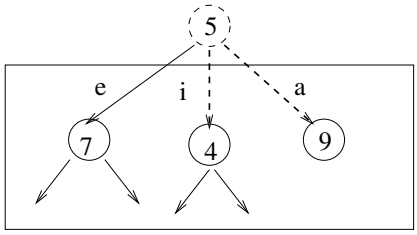


Fig. 2. Node 5 Revisited.

Figure 2 shows the next-nodes of **node 5**. Figure 3(a) shows the storage of this part of the state machine. The outgoing edges of this node are: **e, i, a**. Consider edge **e**. This edge points to **node 7** and so its fan-out is stored. The next-node offset of **e**, NS_Offset_7, points to **node 7**. The rule offset of **e**, Rule_Offset_7, points to a list of matched strings. The entire state machine is stored contiguously in memory and so we use offsets instead of pointers. The address of next-node is computed by adding the next-node offset to an a-priori known base address.

Before we proceed, a terminology clarification. Hereafter, a reference to edge information refers to: next-node fan-out, next-node offset and rule-offset. For example, edge information stored for **edge e** in the above example is **2, NS_Offset_7, Rule_Offset_7**. The string matching algorithm at **node 5** is as follows. The incoming byte is compared with its edges namely, **e, i, a**. In case of a match with any edge, the corresponding edge information is read.

Note that **node 5** is contiguously stored in memory. With contiguous storage, edges of a node can be traversed with an 8 B stride. With cache lines spanning multiples of 8 B, this storage can exploit locality across edges. Additionally, contiguous allocation also opens up avenues for re-arrangement within a node.

This proposed storage is different from array based structures used in earlier works [10, 12, 19, 21]. In an array based structure, the size of each node is fixed irrespective of its fan-out. In contrast, in this storage, the node size is linearly dependent on fan-out[1]. Importantly, 70% of nodes have fan-out of **20** or less, and so we are able to store the state machine in 5.1 MB (from 44 MB with base implementation).

[1]Fan-out X 8 B

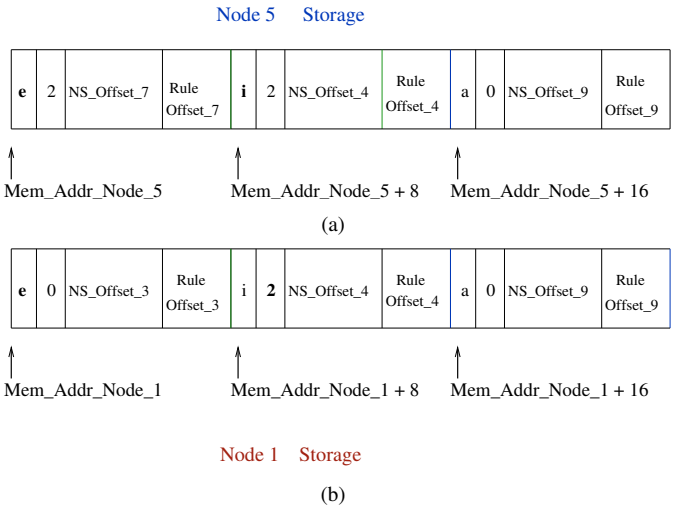


Fig. 3. Our Proposed Storage

However, there are drawbacks. There are 30% of nodes with fan-out of **20** or greater. This is non negligible. So we investigate approaches to reduce this overhead.

A. Fan-out Reduction

We optimize failure edges to reduce the fan-out of nodes. A failure edge, as explained in Section II, indicates a suffix in a string that matches the prefix of another string. For example, edge **i** from **node 5** to **node 4** is due to suffix 'hi'. We observe for Snort database that 93% of edges correspond to failure edges, and this consequently increases the state machine size. Consider the storage of **Node 5** and **Node 1** (refer to Figure 3).

In the storage for **node 5**, **i** and **a** are failure edges that point to **nodes 4, 9** respectively. Also, these edges are non failure edges of **node 1** with exactly the same edge information as well. In other words, edges of **node 1** are replicated in **node 5**. Additionally, this happens for all nodes with failure edges pointing to **nodes 4, 9**. In case of a high fan-out for **node 1**, the impact of replication will be more severe.

Note that failure edges of **node 5** can also be traversed by jumping to **node 1** from **node 5**. **Node 1** can be viewed as a failure-pointer of **node 5**. The advantage with this traversal is that failure edges are not replicated in **node 5**. A unique character, **uchar**, is used to indicate the presence of a failure-pointer in the node. This unique character is not present in the Snort database[2] and hence it is used. Figure 4 shows the modified storage for **node 5**. The edge information corresponding to **uchar** is the edge information of **node 1**. Now the traversal is as follows. The incoming byte is first compared with **e** and in case it doesn't match, the existence of **uchar** is checked in **node 5**. If a failure-pointer exists (and

[2]We have also implemented an alternative mechanism in using unique bit patterns in case all characters are used.

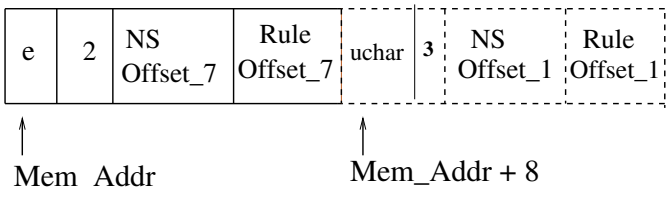


Fig. 4. Fan-out Reduction for Node 5.

it does in this case), it is traversed and the above outlined steps are again repeated for **node 1** . Note that each node has at most one failure-pointer. Failure edges to root-node are dealt separately as well and are explained in the following sub-section.

B. Root-Node Storage

The very high fan-out (**103**) of the root-node together with its very high access frequency motivates us to explore a different structure for root-node. We use an array structure. Each element in the array stores the edge information of the corresponding edge. In the considered example, the edge, **s**, stores the edge information of **node 2** (refer to Figure 5). So

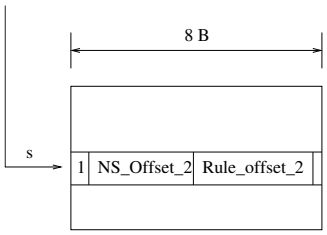


Fig. 5. Root Node Storage.

this representation needs 256X8 = **2048 B**. The traversal using this root-node structure consists of indexing with the incoming byte into the array structure. Since the root-node is frequently accessed and it is only few KBs, we store it on chip. The other-level nodes are stored in an off chip SRAM.

An interesting case arises with failure edges. Consider for example the failure edge, **s**, from **node 7** to **node 2**. Note that the optimized storage stores failure-pointers instead of failure edge, and in this case all these failure-pointers point to the root-node. The failure-pointers to root-node themselves are replicated. So we remove these failure-pointers to the root-node, and index into the root-node if there are no matching edges.

C. Area Comparison

Figure 6 shows the area needed for storing the state machine using various proposals. The area results are obtained using CACTI [20] for SRAMs. Baseline refers to the state machine built using the base implementation of 256 next-node pointers and BS-FSM[3] refers to the state machine built using [19].

[3]Bit-Split FSM - the name used in [19]

The base implementation requires two order of magnitude of

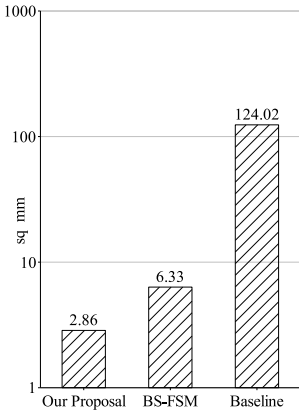


Fig. 6. Area Comparison for Various Proposals.

additional area in comparison to the other two proposals. So it is not relatively area efficient. In comparing BS-FSM and **Our Proposal**, we observe a **2.2X** reduction in area for **Our Proposal**. The BS-FSM requires **6.33 mm**^{2} (2435 KB) for storing the state machine, while our proposed storage needs **2.86 mm**^{2} (1034 KB).

V. Improving the Performance Efficiency

Having investigated techniques to improve the area efficiency, we now present mechanisms to improve the performance efficiency. In this section, we first present enhancements for accelerating other-level nodes.

A. Rearranging Edges

The number of memory accesses to read all edges of a node is dependent on the fan-out. For example, two memory accesses are needed for reading all the edges (**e, uchar**) of **node 5**. This is again a performance penalty and we investigate ways to reduce this penalty.

The traversal of a node can be split into two phases namely, edge scanning, comparing the incoming byte with all edges; and reading the edge information. Edge scanning needs to be performed for all incoming bytes, and so this is a potential performance penalty. We investigate a technique to reduce it. If all edges of a node are stored contiguously, then fewer

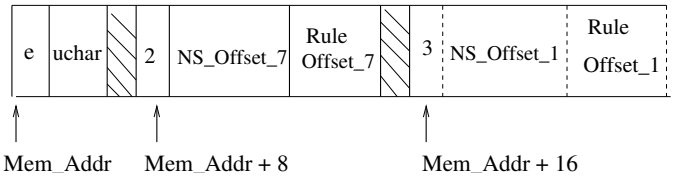


Fig. 7. Fine Tuning for node 5.

memory accesses are needed. Consider **node 5** (refer to Figure 4), we re-arrange this node so that edges are grouped together (refer to Figure 7). With this re-arrangement, all edges of **node 5** are read in just one memory access. In case the incoming byte matches any of these edges, the corresponding edge information is obtained with another memory access. In our simulations for first phase, we perform 8 B memory read operations, so only **fan-out mod 8** memory accesses are needed.

Additionally, note that comparison of edges with the incoming byte proceeds first with **e** and then with **uchar**. This operation can be parallelized with a vector comparator, which can further reduce the computational overhead. We assume the support of an 8 B vector equal-to comparison operation in our simulated architecture. An 8 B vector equal-to comparator primarily consists of 64 AND gates.

Note that this re-arrangement would not guarantee memory alignment for edge information. So we pad the data-structure so that memory accesses are aligned to the nearest 8 B boundary (as shown in Figure 7). Further, note that the area results presented in Section IV are obtained with this aligned storage. Algorithm 1 summarizes the algorithm used for the state machine traversal using our proposed storage.

Algorithm 1 Traversal Using the Proposed Structure.

```
1:  j ← 0
2:  while j < fanout do
3:     Get Edges {8 B Memory Read}
4:     if Edge exists in Incoming Byte {Edge scanning} then
5:        Get Edge Info {Read the Edge Information}
6:        if Rule_Offset ≠ 0 then
7:           Alert {Signal System Alert}
8:        end if
9:     end if
10:    j ← j + 8
11: end while
12: if Failure Ptr Exists then
13:    Get Fail Node Edge Info {Read the Fail Node Info}
14:    if Rule Offset ≠ 0 then
15:       Alert {Signal System Alert}
16:    end if
17: else
18:    Root Node Access
19: end if
```

B. Accelerating Root-Node Accesses

We observe that, for the various traces evaluated, up-to 93% of incoming bytes access the root-node. The root-node can be accessed in two ways. Let **s, h, e, h** be incoming bytes to the state machine depicted in Figure 1. The first byte, **s**, accesses the root-node directly. The subsequent 2 bytes (**h, e**) result in going down the state machine and onto **node 7**. Now the final byte, **h**, scans the edge of **node 7** and on not finding a match, jumps to **node 3** (its failure-pointer as failure edges

are optimized out). Since there are no matches in the failure-pointer as well, then the root-node is accessed with this byte (**h**). Note that the first byte, **s**, accesses the root-node directly while the final byte, **h**, accesses the root-node indirectly. These indirect root-node accesses can potentially be accelerated if it is possible to avoid unnecessary accesses (unnecessary in hindsight) to **node 7, node 3**.

Figure 8 shows the split of root-node accesses as either: direct, indirectly from first-level nodes (**nodes 1, 2**), or indirectly from lower-level nodes (levels lower than first-level). A significant percentage (at least 50.14%) of incoming bytes access the root-node indirectly. We observe that up-to 42% (UPC trace) of root-node accesses, access the root-node indirectly from the first-level. So we concentrate on accelerating indirect root-node accesses arising from first-level nodes.

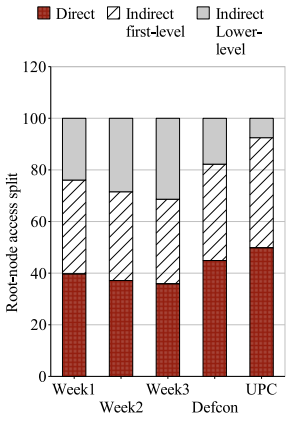


Fig. 8. Root Node Access Split

As discussed in Section IV-B, the root-node is an array of edge information. In addition to this edge information, we store a bit-vector of 256 bits alongside the edge information. A bit in the bit-vector is set for all outgoing edges of that node. Figure 9 shows the storage of the outgoing edge **h** of the root-node. The bits in the bit vector are set for all outgoing edges of **node 1** , namely, **a, e, i**.

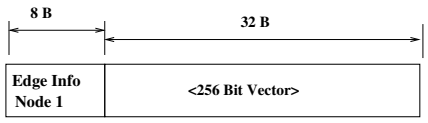


Fig. 9. Storage of the Outgoing Edge e.

The traversal using this enhancement is as follows. Let **h, a** be incoming bytes. The first byte, **h**, results in a direct root-node memory access. The second byte, **a**, also reads the root-node memory and checks if the bit in the bit-vector of the edge **h** (also the previous byte) is set. If it is set, then the other-level node structure is accessed as previously. If the bit is not set, then this is a root-node access and the corresponding edge information is read. In this way, we completely eliminate indirect root-node accesses from the first-level. Note that in order to access the bit-vector we only need the current and previous byte.

The edge information that is read on a root-node access is used to process the next byte. This can be viewed as *root-node edge information*. However, if the next byte also accesses the root-node then the *root-node edge information* is not needed. We note that there are up-to 14 consecutive bytes that access in this manner[4]. For these root-node accessing bytes we only need to check if the bit is set (indicating a root-node access) in the bit-vector. We further accelerate these accesses by pipelining this operation. The stages of the pipeline are the various steps involved, namely,

- *Address Generation*. Computing the memory address of the bit-vector. The current byte and the previous byte are used to compute this index.
- *Root-Node Memory Read*. Reading the bit-vector from root-node memory.
- *Bit-set Check*. Checking if the corresponding bit is set in the bit-vector.

The pipeline latency is that of accessing the root-node memory and is 3 clock-cycles (obtained from CACTI [20]). The pipeline is flushed in case the bit is set, then the other-level node structure needs to be accessed.

Thus for consecutive bytes directly accessing the root-node, the throughput will be 3 clock-cycles per byte. Note that the bit-vector enhancement requires an additional 8 KB (32 B X 256) of root-node memory.

C. Hardware Architecture

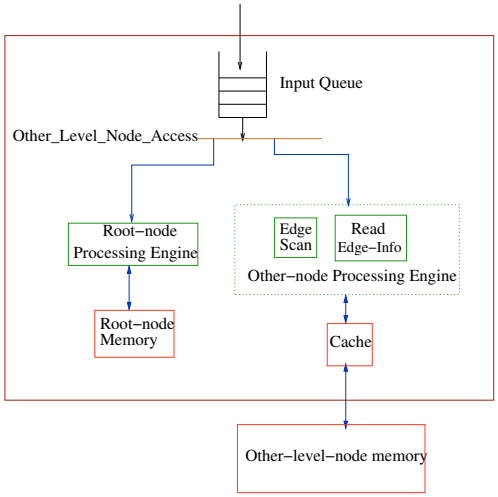


Fig. 10. Proposed Hardware Architecture

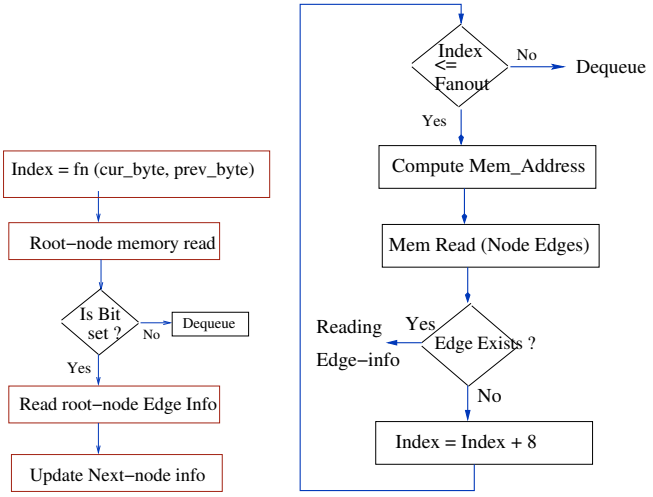
The hardware is customized for traversal using our proposed storage. The hardware architecture (refer to Figure 10) consists

[4]For the UPC trace

of a *Root-node processing engine*, an *Other-nodes processing engine* and the state machine memory. The flag, *Other-level-node-access*, determines the engine to be used for processing the current byte. If this flag is set, then the *Other-nodes processing engine* processes the byte. This flag is accordingly updated for every byte after completing its processing.

Incoming bytes from the network are buffered in the input queue and dequeued after their processing is complete. The *Root-node processing engine* processes a byte if the *Other-level-node-access* flag is not set. Figure 11(a) shows the processing flow-chart for the *Root-node processing engine*. This engine performs two functions, namely, checking if the bit in the bit-vector is set, and reading the *root-node edge information*. The bit set check is performed by the 3 staged pipeline unit as discussed in the previous section (Section V-B). Note that the first 3 steps of root-node processing are also the 3 pipeline stages. The hardware logic needed for this unit are: an 8 B Equal-to comparator and a shift-and-add logic block. In case the bit is set, then the *root-node edge information* is read, which needs a shift-and-add and an AND gate. The *Other-level-node-access* flag is also set in this case.

The *Other-nodes processing engine* traverses the state machine using Algorithm 1. The traversal operations consists of: scanning all the edges of a node, and reading the associated edge information of the matching edge. So we split this engine into these operations (refer to Figure 11). In edge scanning,



(a) Root-node Processing Engine (b) Other node Processing Engine - Edge Scanning

Fig. 11. Processing Flow-charts

all edges of a node are read, eight at a time, and compared (vector comparison) with the current byte. This is iterated over all edges until a matching edge is obtained. If a matching edge or a failure-pointer exists, then the associated edge information is read. Otherwise, the root-node is accessed and *Other-level-node-access* is reset.

For edge scanning the hardware needed is: an 8 B vector equal-to comparator, a shift-and-add, and a less-than-equal-

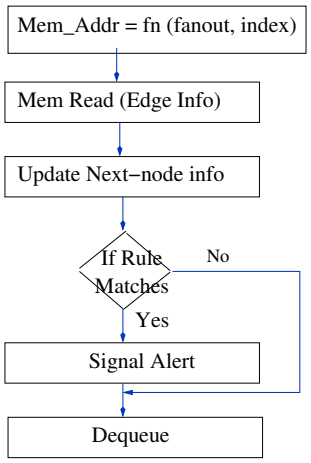


Fig. 12. Edge Information Processing

to comparator. While for reading the edge information, the hardware required is a shift-and-add and an equal-to comparator. Note that identical steps (edge scanning and reading edge information) are also followed for failure-pointers. We assume that each of the arithmetic processing blocks need 1 clock-cycle and *if comparisons* need 2 clock-cycles.

VI. Simulation Methodology

We evaluate the performance of our proposed architecture and compare it with the base Aho-Corasick and BS-FSM [13, 14, 19]. We have used 5 network traces in our evaluation, including an in-house university trace. We use 3 publicly available traces from Lincoln Labs [11], and a trace from Defcon-8 Capture the flag (CTF) game [5]. We have inspected TCP, ICMP and UDP packets from these traces.

Table I summarizes the traces used. Week 1, 2 and 3 data-

Data-sets	Mean Packet Size (B)	Num Packets (M)
Week 1	344.3	6.07
Week 2	160.51	13.18
Week 3	200.01	14.91
Defcon	71.9	15.64
UPC	535.87	15.89

TABLE I
Summary of Traces used in Evaluation.

sets refers to the respective Lincoln Labs 1999 week traces, and these are actually five day aggregates. The in-house trace (referred to as UPC) was collected from the university router on November 7, 2007 at 18:00 hrs.

We have used the Snort database released on September-2007, which contains 23,653 strings. We also later report results for the April-2010 Snort database containing 40,678 strings.

We use **average number of clock-cycles per incoming**

byte as the metric for performance comparison. This is computed by dividing the total number of clock-cycles by the total number of bytes. Total number of clock-cycles is the sum of **total processing time** and **total memory access time**. **Total processing time** comprises of: edge scanning, reading edge information and root-node processing. Note that processing times for edge scanning, edge information and root-nodes processing are obtained as explained in Section V-C. The **total memory access time** is obtained from the trace driven cache simulator [7], which was modified to model cache access times and processing times. The cache miss penalty is the other-level node memory access latency and is obtained from CACTI [20]. The cache hit time is 2 clock-cycles (also from CACTI). The core frequency is assumed to be 3 GHz.

We have compared our proposed architecture with the base Aho-Corasick and with bit-split FSM (BS-FSM) based approaches [13, 14, 19]. BS-FSM uses state machines constructed from bits instead of bytes. There are multiple state machines, each constructed using a set of bits from the byte. Further, they observe that 2 bits, and therefore 4 (8b/2b) state machines is the optimal point. Importantly, these state machines are traversed in parallel and we simulate it using 4 individual cores for the 4 state machines. We assume that the 4 cores also have their private caches, and SRAM (which stores the state machine) has 4 memory banks. Each of these cores emit partial string match vectors for every byte and an intersection of these vectors indicates a string match. This intersection is a synchronization operation across the cores and is done by another core. Further, this intersection is done using a vector comparator of vector-length equal to **23,653**[5]. The four partial match vector intersection is performed using 3 vector AND operations each of 1 clock-cycle latency. Piyachon et al [13, 14] propose memory-efficient storage for these bit vectors. However, these techniques need additional memory operations. So effectively we have simulated and compared with the upper bound of BS-FSM based approaches. Figure 13 shows the architecture used for evaluating BS-FSM [19]. We

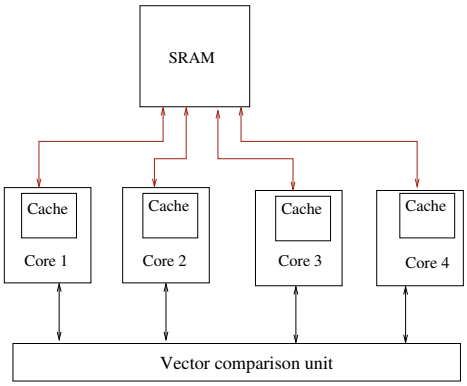


Fig. 13. Architecture of BS-FSM.

obtain the total processing times for these comparison schemes by executing their respective traversal algorithm on an in-order single-issue processor (using Simplescalar [3]). Table II shows processing clock-cycles needed per byte.

Schemes	Clock-cycles per B
Base Aho-Corasick	11.93
BS-FSM [19]	18.82

TABLE II
PROCESSING CLOCK-CYCLES FOR COMPARISON SCHEMES.

We use a 16k direct-mapped cache configuration for the cache in *Other-node-processing engine*. This configuration is used, since we observe no significant performance improvement in larger cache size and/or associativity. In the case of BS-FSM, each of the 4 cores have 64k 4-way associative cache. The base Aho-Corasick scheme also uses a 64k 4-way cache. Cache lines are 16 B and an *lru* replacement is used. We use these cache configurations in our further discussion.

In case there is a string match, then the system needs to be alerted and the packets are logged into system log. This logging is performed by a dedicated core, which is the case for all the evaluated proposals.

In our evaluation, we have used SRAM memory **double** the state machine memory size. This is driven by the fact that the SRAM needs to be sufficiently provisioned for the exponentially growing database. Given that Snort database has doubled over the last 30 months, we have projected for the future. This **provisioning** is done for all schemes that are evaluated.

VII. RESULTS

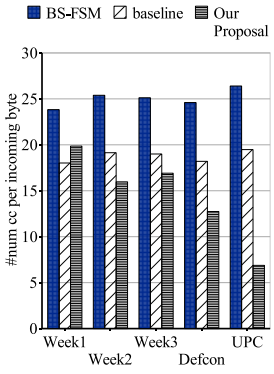


Fig. 14. Performance Comparison.

Figure 14 shows the performance of various proposals for all traces. We observe that **Our proposal** outperforms other schemes in 4 of the 5 traces. For the UPC trace, **Our proposal** needs only **6.88 clock-cycles** per byte. On the other hand,

[5]number of strings in Snort database

the baseline and BS-FSM need **19.48 and 26.4 clock-cycles** respectively. For this trace, we obtain a performance improvement of **73%** compared to BS-FSM. A similar performance behaviour is observed in Defcon trace with **48%** performance improvements over BS-FSM. For these traces, we observe that more than 80% of bytes access the root-node, further, more than 80% result in direct root-node accesses. The proposed pipelined architecture accelerates these multiple consecutive direct root-node accesses thus providing performance benefits.

It is interesting to note that the Week1 trace has a different performance behaviour. The baseline actually gives better performance than **our proposal**. Further, there is a 3X degradation for **our proposal** in comparison to the UPC trace. This behaviour can be explained as follows. Only 48% of bytes in Week1 trace result in direct root-node accesses, in contrast to 93% of bytes for UPC trace. Additionally, more than 30% of the byte content in Week1 trace is **0**, and there exists a string of **20 consecutive 0**s in Snort database. This results in frequent traversal of this string's node at level 20 and its predecessor at level 19 - this node is the failure-pointer of the level 20 node. We observe that close to 30% of bytes in this trace access only these nodes. The traversal of failure-pointer incurs a performance overhead in our proposal, namely, of an additional memory access and the associated edge information processing. This results in performance degradation for our proposed architecture. We also observe a similar, albeit less frequent, behaviour in Week2 and Week3 traces as well.

Note that though we have used throughput metric as **average number of clock-cycles per B (cpB)**, we can also obtain throughput in terms of **Gbps** by simple arithmetics (processor clock frequency/(cpB/8)). For example, the throughput of **Our proposal** with UPC trace is **3.4 Gbps**.

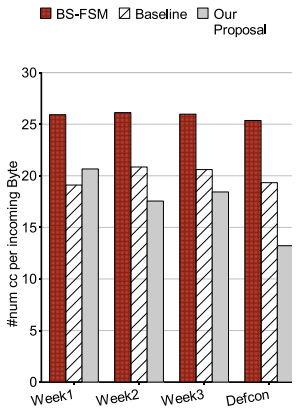


Fig. 15. Performance Comparison for April-2010 Rulesets.

Figure 15 shows the performance results for April-2010 Snort rulesets. We again observe a similar behaviour, **Our Proposal** outperforms both BS-FSM and baseline in 3 of the 4 traces[6]. For example, in Week2 trace **Our Proposal** needs **17.55** cc per byte while baseline and BS-FSM need **20.85 and 26.12** cc per byte respectively. A similar performance behaviour is also observed in Defcon and Week3 traces. However, for the Week1 trace **Our Proposal** outperforms BS-FSM but there is a performance degradation wrt baseline (20.66 vs 19.1 cc per byte).

[6]Due to time constraints results could not be obtained for the UPC trace

It is interesting to observe that BS-FSM, even with additional hardware resources (4 additional cores, 4 banked memory, and a 23,653 long vector unit), does not yields significant performance improvement. The performance suffers due to the high miss rate of the caches in BS-FSM. The caches in BS-FSM have an interesting behaviour. If, at least, one of the 4 cores incurs a cache miss then the processing stalls, since an intersection of partial match vectors needs to be done for every byte. We observe that 50% of bytes (in UPC trace) incur these misses, thereby lowering the performance drastically.

The hardware needed for traversal using our storage can be summarized to consists of: an 8-B vector EQUAL-TO comparator (64 AND gates), 3 shift-and-add operators, 3 mask operations, an adder, and 2 less-than-equal-to comparators. These are not complex logic blocks and can be incorporated without significantly increasing the chip complexity. Note that our proposed state machine storage gives **43X** area improvement in comparison to the baseline, and over **2x** in comparison to BS-FSM. Additionally, these schemes also need 64k 4-way assoc cache (in case of BS-FSM there are 4 such caches), while **our proposal** needs a 16k direct-mapped cache. In case of BS-FSM, the on-chip area also consists that of 4 additional cores and a very long vector comparison unit. Given the area savings obtained, the chip area for the proposed logic blocks will most likely be offset by the area gains.

VIII. Conclusion

In this work we have investigated various mechanisms to improve the area and performance efficiency of an IDS. In our proposal for improving the area efficiency, we have investigated a novel type of storage for storing the state machine. Using this novel storage, we reduce the area needed by 2X magnitude in comparison to BS-FSM [19]. We investigate mechanisms to improve the performance efficiency of the IDS. Most notably, we observe that consecutive bytes in the trace directly access the root-node. Based on this observation, we propose a pipelined architecture for processing these multiple consecutive bytes. We compare the performance of our proposed architecture with BS-FSM based approaches and baseline. Our performance results indicate that our proposed architecture outperforms BS-FSM based approaches [13, 14, 19].

One way to further improve the efficiency of our proposed architecture is to dynamically determine frequently accessed nodes (similar to the root-node) and modify the layout at run time. It will also be interesting to study the behaviour of this architecture under performance attacks.

ACKNOWLEDGMENTS

This work has been supported by the Spanish Ministry of Education and Science under grants TIN2007-61763, TIN2010-18368 and by the AGAUR, Generalitat de Catalunya under grant 2009 SGR 1250.

We would like to thank Josep Sole Pareta, Josep Sanjuas and Pere Barlet for providing logistic support and access to UPC university traces. Additionally, we also would like to thank numerous anonymous reviewers for providing insightful comments on earlier versions of this work.

REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. Communications of the ACM (CACM)1975.

[2] M. Becchi and S. Cadambi. Memory-Efficient Regular Expression Search Using State Merging. In Proceedings of the IEEE Infocom 2007.

[3] D. Burger and T. M. Austin. The Simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin 1997.

[4] Y. H. Cho and W. H. Mangione-Smith. Deep Packet Filter with Dedicated Logic and Read only Memories. In Proceedings of the IEEE Field Programmable Custom Computing Machines (FCCM) 2002.

[5] Defcon Hacking Conference, Capture the Flag (CTF) traces available at http://cctf.shmoo.com.

[6] 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.

[7] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator, http://www.cs.wisc.edu/˜markhill/DineroIV.

[8] Intel Corporation, Intel IXP 2400 Network Processor Hardware Reference Manual, Revision 7 2003.

[9] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In Proceedings of the ACM SIGCOMM 2006.

[10] C. Lin, Y. Tai, and S. Chang. Optimization of Pattern Matching Algorithm for Memory Based Architecture. In Proceedings of the ACM/IEEE Architecture Network and Communication Systems (ANCS) 2007.

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

[12] P. Piyachon and Y. Luo. Efficient Memory Utilization on Network Processors for Deep Packet Inspection. In Proceedings of the ACM/IEEE Architecture Network and Communication Systems (ANCS) 2006.

[13] P. Piyachon and Y. Luo. Compact State Machines for High Performance Pattern Matching. In Proceedings of the ACM Design Automaton Conference (DAC) 2007.

[14] P. Piyachon and Y. Luo. Design of a High Performance Pattern Matching Engine Through Compact Deterministic Finite Automata. In Proceedings of the ACM Design Automaton Conference (DAC) 2008.

[15] M. Roesch. SNORT - Lightweight Intrusion Detection for Networks. In Proceedings of LISA-99: USENIX 13th Systems Administration Conference 1999.

[16] I. Sourdis and D. Pnevmatikatos. Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. In Proceedings of the IEEE Field Programmable Custom Computing Machines (FCCM) 2004.

[17] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos, and S. Vassiliadis. Packet Prefiltering for Network Intrusion Detection. In Proceedings of the ACM/IEEE Architecture Network and Communication Systems (ANCS) 2006.

[18] R. Smith, C. Estan, and S. Jha. XFA: Faster Signature Matching with Extended Automata. XFA: Faster Signature Matching with Extended Automata. In Proceedings of the IEEE Symposium on Security and Privacy 2008.

[19] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA) 2005.

[20] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HP-2008-20, HP Labs 2008.

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

[22] F. Yu, R. H. Katz, and T. Lakshmanan. Gigabit Rate Packet Pattern-Matching Using TCAM. In Proceedings of the IEEE International Conference on Network Protocols (ICNP) 2004.