High-performance String Search Engine for Network Security using Random-Access-Memories

Young H. Cho, *Member, IEEE,* and William H. Mangione-Smith, *Member, IEEE*

***Abstract*— Due to affordable broadband internet access, more people are using the computer network to do their everyday activities than ever. Due to minimal security measures provided by the service providers, most internet users today are vulnerable to several malicious attacks through web sites and e-mails. In 2003, it has been estimated that computer network worms and virus caused the loss of over $55B. Unless the security systems use more advanced techniques to scan all the aspects of the packets, the damage will likely to increase in the future. While there are a few intrusion detection system running on general purpose processors, they lack the processing power to monitor gigabit networks. There have been a few research projects that use reconfigurable devices to support these higher speed networks. However, unlike the software solutions, the reconfigurable implementations require longer time to apply the updates to the signature set. We present a high performance pattern matching co-processor architecture that can be used to monitor and identify a large number of intrusion signature without a need for hardware reconfiguration. The design consists of a bank of pattern matchers that are used to implement a highly concurrent filter. The pattern matchers can be programmed to match multiple patterns of various lengths, and are able to leverage the existing databases of threat signatures. We have been able to program the filters to match all the payload patterns defined in the widely used Snort network intrusion detection system at a rate above 7 Gbps, with memory space left to accommodate threat signatures that become available in the future.**

***Index Terms*— Network, Security, CAM, String, Search**

I. Introduction

Most firewalls today are equipped to examine the packet headers only. Therefore, application layer network attacks such as e-mail attachments can slip through the security systems undetected. While e-mail lends itself to store and scan techniques, such as those developed by anti-virus companies, other applications (e.g. databases) may not.

One effective security measures for such attack is deep packet inspection [1]. Deep packet inspection not only examines the packet headers but also the payload data. Therefore, a security system that incorporates deep packet inspection offers better protection from attacks than traditional firewalls. It is evident that traditional firewalls that in much use today have not been effective in differentiating network packets containing worms from normal e-mails. However, deep packet inspection system, such as Snort [2]–[4], can be configured to detect several different worms by searching for specific patterns in the network packet payload.

This research work was supported under NSF grant #CCR-0220100.

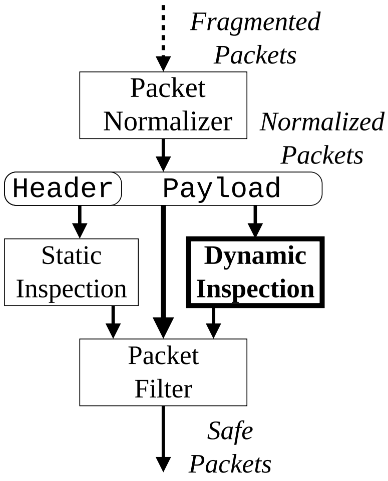


Fig. 1. Deep Packet Inspection

A. Deep Packet Inspection

The Internet traffic is made of streams of fragmented packets with different source and destination addresses. Since attack can span more than one packet of a stream, every stream needs to be reassembled before applying the deep packet inspection. There are also some class of attacks that use unconventional protocol features to confuse and avoid the intrusion detection system. One such attack uses overlapping fragmented IP packets. Such must be eliminated by normalising the packets. Packet normalization produces consistantly clean network traffic without abnormalities [5]. Figure 1 shows the steps of an effective deep packet inspection.

Most of the currently available deep packet inspection systems use one or more general purpose processors running signature-based filtering software. Although these software systems can be easily reconfigured to detect new attacks, the underlying processor are not powerful enough to sustain acceptable filtering rate on gigabit (and above) networks. For example Snort, one of the most widely used software system, when configured with 500 real string patterns can only sustain a bandwidth less than 50 Mbps on a dual 1 Ghz Pentium 3 system.

Since the payload data is under the control of the user application, all the patterns must be compared at every byte of the payload during the search process. Therefore, as the number of patterns in the software system increases, the filtering process needs more processing power. We refer to this pattern matching task as the dynamic inspection.

This exhaustive search process on general purpose processor is expensive and the current software solutions are impractical for networks over 1 Gbps. Therefore, we have developed a specialized pattern matching co-processor for the dynamic pattern search.

In the following section, we briefly discuss how recent developments on reconfigurable hardware allows deep packet filtering on high bandwidth network. Then we present our novel architecture for 1+ gigabit networks in section 3. In section 4, we describe our initial pattern matcher capable of detecting all the patterns defined in current Snort rule set. Since the architecture does not require hardware reconfiguration, our intial performance measurements are based on the ASIC design using 0.18μm library. We expand our discussion in section 5 by suggesting ways to utilize the network traffic statistics to increase the engine performance. Finally, in section 6, we make our concluding statement.

II. Related Work

Due to lack of performance in software systems, several researchers have looked into developing special pattern matching units in field programmable gate array (FPGA) devices.

Sidhu and Prasanna mapped Non-deterministic Finite Automata (NFA) for regular expression into FPGA to perform fast pattern matching [6]. Subsequently, Franklin and Hutchings implemented a pattern search engine in JHDL, based on a subset of Snort IDS rules [7]. At around the same time, we developed an FPGA filter that used 8-bit decoders to build 3.2 Gbps pattern match engine on FPGA [3]. Based on the above concepts, Sourdis deepened the pipeline to increase the filtering rate to 10 Gbps [8].

Our follow-up work and a similar JHDL based design by Clark and Baker made contribution in reducing the size of the design by eliminating duplicate logic [4], [9], [10]. Such improvement allowed the decoder design to fit into a single FPGA with performance of several gigabits per second

Gokale et al. of Los Alamos National Laboratory implemented a fast re-programmable pattern search system using content addressable memories (CAM) [11]. Although such system does not require reconfiguration of FPGA, the low performance of CAM limits the usefulness as well as the number of mappable rules.

Dharmapurikarup et al. from Washington University presented an approximate method using Bloom filters [12]. They detect the patterns at 600 Mbps with number of false positives which is dependent on the number of rules as well as size of the alotted memory. Their approach uses hashing, and ultimately requires a secondary exact string comparison process to detect false positives [13].

Our latest FPGA implementation that uses a combination of 8-bit decoders and read-only-memory to reduce the amount of discrete gates by store partial information in the memory. The logic savings is achieved by using the decoders to generate the address for the partial pattern entry in a ROM. By balancing the use of the discrete gates and memory, this yields the highest performance per gate [4], [14].

Unlike the software solutions, many of the FPGA implementations performs at 1+ gigabit per second filtering rate. However, other than the CAM implementation [11] and the bloom filter design [12], the FPGA design compilation and reconfiguration time can be in the order of minutes to days. Such delay in reconfiguration may not acceptable as new worms are released to the network in higher frequency.

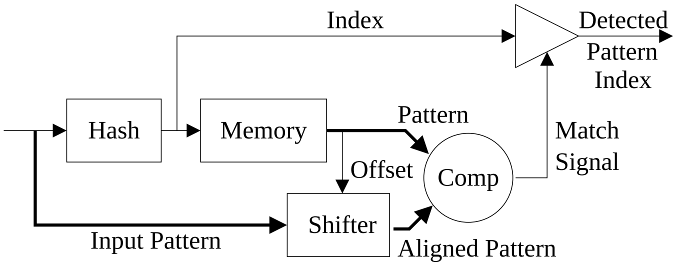


Fig. 2. Pattern Detection Module

III. Architecture

A Snort rule contains information to search through all layers of network packets to detect a particular attack [2]. The most computationally intensive phase of the detection process is an exhaustive string search on the packet payload. We present a compact and programmable pattern search co-processor for multi-gigabit per second network.

A. Pattern Detection Module

The basic pattern detection module (PDM) is shown in figure 2. The function of the pattern detection module is to efficiently detect segments of pattern using programmable hash functions followed by discrete string comparison.

At every clock cycle the input pattern is hashed to generate an index. The index is then used as an address of a memory where the corresponding pattern is stored. The retrieved pattern from the memory is then compared with the input pattern to determine whether the pattern is an exact match. When there is a match, the index can be forwarded with an unique identifier for the pattern.

We use parametrized and cascaded hardware so that the length of the patterns are not fixed. Therefore, the maximum length of the input bytes that is used to generate the hashed index is the minimum length of the patterns detectable by a single PDM. Moreover, the maximum range of the hashed index determines the maximum entries that can be stored in the memory. For instance, if two byes of the input pattern are hashed to generate the index, the PDM can be configured to detect maximum of 65,536 patterns with the minimum length of two bytes.

1) Hashed Index: Hashing the substrings in a static position places a constraint on which patterns can be detected by one PDM. If the first two bytes of all the patterns are used for generating the index, some would have the same hash value and could not be stored in the same PDM. For higher resource utilization, we allow the index to be generated by any substring of the pattern.

In practice, each pattern consists of more than one unique substring. By allowing the hash function to start at different

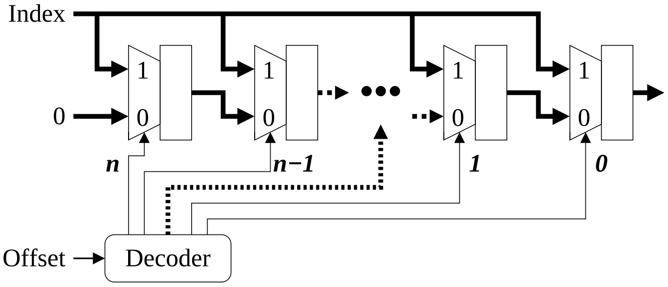


Fig. 3. Switched Pipeline

byte offsets of the pattern, the PDM memory utilization can be improved. Therefore, the byte offset data is stored with the pattern information in the memory. Using the offset and the pattern length, the input pattern is shifted and compared against the target pattern as shown in figure 2.

Since the index is generated from a substring of the pattern at a different offset, the timing of the identification index output may not indicate the starting byte of the pattern. By using the offset value with the switched pipeline as the one shown in figure 3, the index output timing can be adjusted to correspond with the start of the pattern.

2) Prioritized Parallel Modules: Some patterns, especially the ones with a small set of unique substrings, cannot be mapped on to the same PDM module because the entries for their hashed indices might be used by another pattern (e.g. pattern that is made of all zeros). Therefore, more than one PDM must run in parallel to detect multiple patterns with equal hashed values.

In order to increase memory utilization, each PDM can have different sized memory and logic based on a range of target patterns. To maintain consistent output timing for the parallel modules, smaller PDMs may need extra stages of pipeline to match with longest PDM.

If the PDMs are configured to examine the same data simultaneously, in most cases, only one PDM will output a valid index for a pattern match. By extending the output bits to indicate its module number, the outputs from the parallel PDMs can be merged to produce one index output.

Depending on the memory content of the PDMs, more than one PDM can output valid indices at a given cycle. Multiple detections occur if one pattern is a substring, starting with the first byte, of another pattern. We refer to such patterns as “overlapping patterns.” When more than one index is detected in the same cycle, it is sufficient to output the index for the longest pattern since it also indicates the detection of the shorter patterns.

The figure 4 represents the parallel PDMs with priority support. Our design use chains of multiplexors to assign the priorities as well as merge the PDM outputs. By storing the longer of any conflicting patterns in the PDM with the higher priority, the system is capable of detecting of all the overlapping patterns.

The above PDM architecture allows the detection of patterns of lengths that are less than or equal to that of the widest memory module from all the PDMs. We refer to such pattern as “short pattern.”

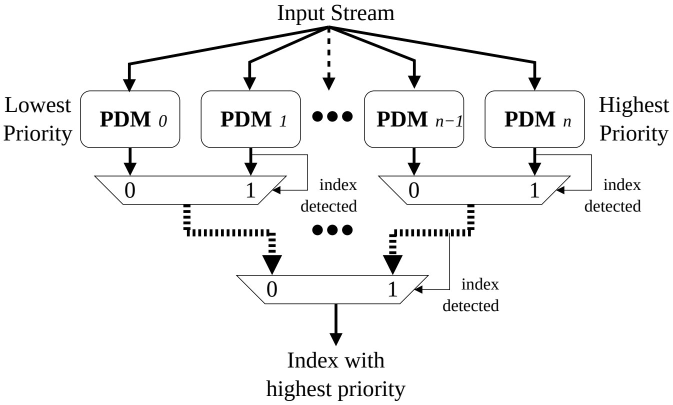


Fig. 4. Parallel PDMs with priority

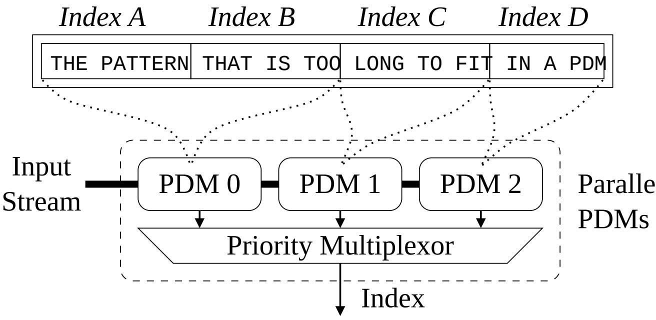


Fig. 5. Divided segments of the long pattern maybe detected by different PDMs

B. Long Pattern State Machine

For applications such as Snort, where some patterns are long, it is not efficient to have the PDM with memory wide enough to store those patterns. In this section, we describe another component that uses PDMs to detect patterns that are longer than the width of PDM memories.

1) Sequence of Segments: Every long pattern can be broken into several short pattern segments. If we match the order and the timing of the segment sequence, we can effectively detect the corresponding long pattern.

As in figure 5, the long pattern is divided into smaller segments that fit in to a specific PDM. These segments are stored in the PDMs along with a flag bit that indicates that it is a segment of a long pattern. The detected indices are forwarded to the long pattern state machine (LPSM).

2) Programmable State Machine: The LPSM examines the sequence of segment indices for the correct ordering and the timing to detect the corresponding long pattern.

As shown in figure 6, the LPSM is consists of the memory and the pipeline similar to that of PDM. Unlike the PDM, the memory only contains information for the current and the next “state”. Each state is expressed as number which is based on the index of the pattern segments detected by the PDMs.

The memory entry in LPSM with the state information is loaded using part of the index identified by the PDMs. The rest of the bits for the index are stored in the memory to verify the current state. The entry also has a type field that indicates whether the current index is the first, the middle, or the last segment of the long pattern. The entry also specifies what the

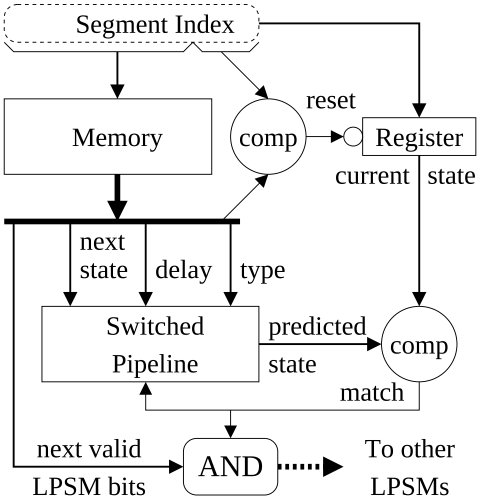


Fig. 6. Long Pattern State Machine

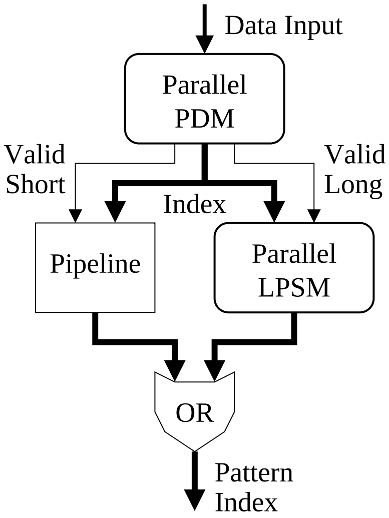


Fig. 7. Block Diagram of Short and Long Pattern Filter

next state is and when it is expected to be detected by the PDMs.

The sequence matching process is only initiated when the type of the current state indicates that it is the start of a long pattern segment. The expected next state is forwarded to the switched pipeline like the one used in PDM to add the appropriate delay. When the next index reaches the end of the pipeline, it is compared with the actual current state to determine whether there was a match.

When the previous next state is an exact match of the current state at the end of the pipeline, the expected next state is forwarded in to the pipeline as before. If the expected next state does not match the current state, this process is terminated without any output. Otherwise, the process continues until the current state is specified as the last segment of the long pattern. Then the last matching index is forwarded as an index for the detected long pattern.

3) Parallel LPSM: Depending on the depth of the LPSM memory and the long pattern indices, more than one entry maybe necessary for the same address. In order to address this, more than one LPSM can run in parallel to detect more than one sequence of states.

In order to interoperate between the LPSMs, the match bit is forwarded to the modules that contain all the corresponding next state for the current state. When any of the LPSM receives the match bit, its expected next state is forwarded to the pipeline regardless of the result in its own comparator.

C. System Integration and Features

Figure 7 is a simplified block diagram of our dynamic deep packet inspection system. As shown in the figure, the short patterns can be detected using only the PDM whereas the long patterns are detected using both the PDM and the LPSM modules.

Unlike the FPGA designs, which required functional circuit changes, this design only requires updating memory values. Therefore, the above system takes much less time to update the inspection rule set than the systems that require changes in hardware.

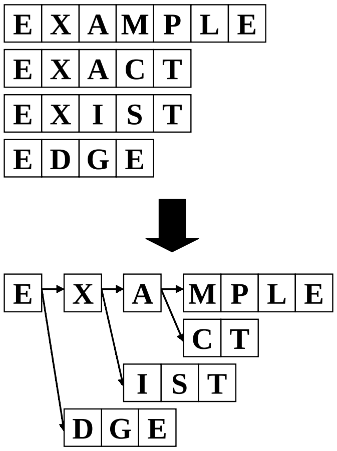


Fig. 8. An example of Aho and Corasick's Keyword Tree: 6 bytes are optimized away.

1) Reusing Memory Entries: Since the multiple index sequences can be tracked by the parallel LPSMs, they can be programmed to reuse pattern segments that appear in more than one pattern. By reusing the pattern segments for more than one pattern, the memory requirement for PDM can be reduced.

Aho and Corasick's keyword tree [15] is used in many efficient software pattern search algorithms, including the Snort IDS [16]. This algorithm is used in the FPGA implementation to reduce the hardware area [4]. We also apply the algorithm to configure the PDM memories.

A keyword tree in figure 8 is one way to store a set of patterns into an optimized tree of common keywords. The conversion not only reduces the amount of required storage, but also narrows the number of potential patterns as the pattern search algorithm traverses the tree.

First, the pattern set must be analyzed to form the keyword trees. Once the keyword trees are generated, its keywords are stored as pattern segments in the PDMs and the edges are stored as the state transitions in the parallel LPSMs. This optimization allows the duplicate pattern segments to be

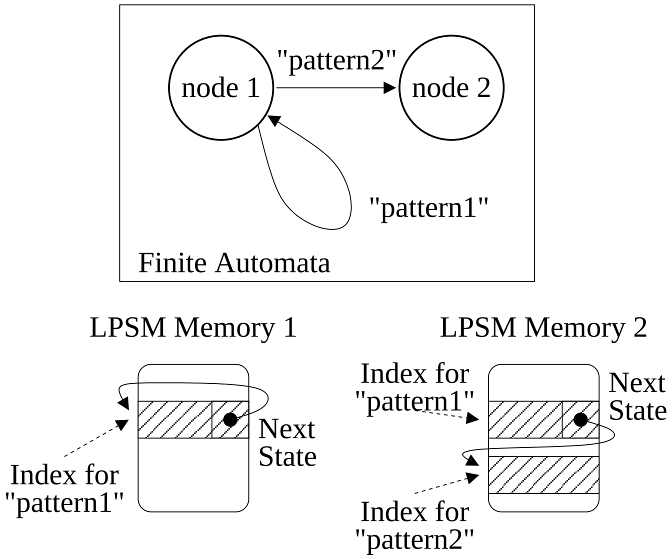


Fig. 9. Regular expression: ("pattern1")+ "pattern2" - one or more instance of "pattern1" followed by "pattern2"

collapsed into a single segment to save PDM memory space.

2) Regular Expression: In addition to keeping track of multiple long patterns, the parallel LPSMs can be programmed to detect regular expressions.

Regular expression can be represented in the form of NFA [6], [7]. Once the NFA representation is formed from regular expression, it can be mapped on to our design. All the inputs to the NFA are recognized by the PDMs while the transition from each node can be mapped on the parallel LPSMs. For the same current state entry, each LPSM can point to the next state that is the next node of the NFA. In similar fashion, the often more compact DFAs can also be mapped in to the design memory [17].

For instance, the node with edges that points to it self and to another node, as shown in figure 9, can be mapped such that the next state stored in one LPSM is the same index as the current state while another LPSM would have next state index that points to the second other node.

IV. Snort Implementation

Snort is one of the most widely used network intrusion detection system (NIDS) that uses deep packet inspection. It is open source software that can be configured with the set of signatures that are used to identify network attacks. In June 2004, the Snort rule set contained 1,729 string patterns that should to be searched dynamically in the network payload. To evaluate the effectiveness of our architecture, we implement the filter based on the architecture to support the entire Snort rule set. Our design contains additional memory space for flexible configuration in the face of new attacks.

A. Hardware Configuration

The dimension of the memories, the number of PDMs, the number of LPSMs, and the hash functions are the architecture parameters. These parameters allow the designer to customize the filter for a given threat profile. Depending on the pattern set, the parameters of the architecture may differ dramatically to optimize the resource utilization. For example, the designer may decide that LPSMs are unnecessary if all the target patterns are short and uniform in length. On the other hand, the designer may choose to have small PDM followed by many parallel LPSMs if the patterns consists of repetitive set of common substrings.

Determining the parameters of the architecture is a complex process which effects the behavior of the system. However, this process is beyond the scope of this paper. Therefore, we attempt to describe one system we have implemented to successfully map the entire Snort rules.

1) PDM Parameters: The length of the patterns range from 1 to 122 bytes in Snort rule set. The contents of the patterns vary from binary sequences to ASCII strings. Therefore, we design the filter to support patterns of various lengths as well as the content. For the pattern set, using different size memories in the PDMs can increase the memory utilization and decrease the logic area. However, we choose to set the dimension of all the PDM to be same to simplify of the design process.

The dimension of the memory in each PDM is 146 bits by 512 entries. The memory is wide enough to store all the information necessary to detect up to 17 bytes long pattern. In our filter, eight of these PDM units are connected in parallel to provide 8-levels of priority.

The filter takes two consecutive input bytes to generate the 9 bit address for the PDM memories. As we mentioned in the architecture description, the minimum pattern length for our filter, therefore, is 2 bytes long. Since single byte pattern can be more efficiently detected using byte decoders, we do not map them on the filter.

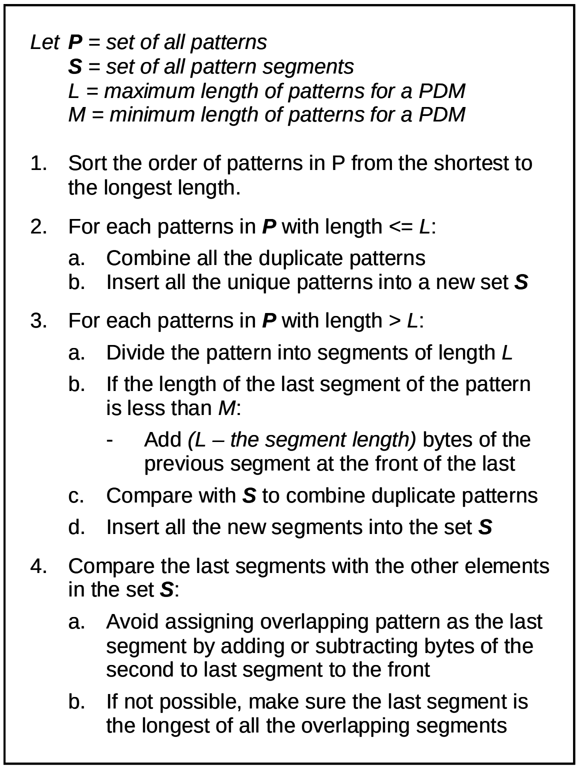
The hash function logic consists of series of multiplexors to independently choose any 9 bits of the 16 bits. The hash logic in each PDM are individually configurable to give more flexibility for the programmer.

2) LPSM Parameters: The design consists of eight units of LPSMs, each with 29 bit by 512 memory entries. Although we found that 256 memory entries per LPSM is sufficient to completely map the entire Snort contents, we use the bigger memory for easier filter programming in the future. Since each LPSM can match different sequence of pattern, the design is capable of reusing one short pattern segment up to eight times.

In order to save memory space, the hashing logic for LPSM uses portion of index bits to load the state information. The index bits 11 through 2 are directly connected to the address of the memory while the rest of the bits are, later, matched with the memory content.

B. Pattern Software

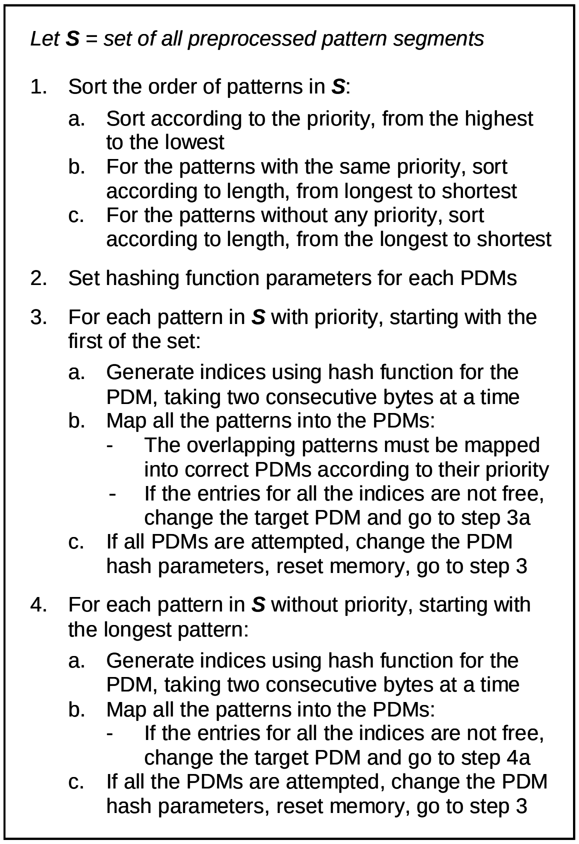
Once the hardware parameters are set, the resulting datapath can be programmed using several different algorithms. Depending on the complexity of the algorithms and the patterns, there can be a big difference in compilation time as well as the program size. In general, reducing the size of the program takes longer compilation time. However, smaller program tend



```
Let P = set of all patterns
    S = set of all pattern segments
    L = maximum length of patterns for a PDM
    M = minimum length of patterns for a PDM

1. Sort the order of patterns in P from the shortest to
   the longest length.
2. For each patterns in P with length <= L:
   a. Combine all the duplicate patterns
   b. Insert all the unique patterns into a new set S
3. For each patterns in P with length > L:
   a. Divide the pattern into segments of length L
   b. If the length of the last segment of the pattern
      is less than M:
      - Add (L – the segment length) bytes of the
        previous segment at the front of the last
   c. Compare with S to combine duplicate patterns
   d. Insert all the new segments into the set S
4. Compare the last segments with the other elements
   in the set S:
   a. Avoid assigning overlapping pattern as the last
      segment by adding or subtracting bytes of the
      second to last segment to the front
   b. If not possible, make sure the last segment is
      the longest of all the overlapping segments
```

Fig. 10. A simple algorithm to divide patterns for PDMs



```
Let S = set of all preprocessed pattern segments

1. Sort the order of patterns in S:
   a. Sort according to the priority, from the highest
      to the lowest
   b. For the patterns with the same priority, sort
      according to length, from longest to shortest
   c. For the patterns without any priority, sort
      according to length, from the longest to shortest
2. Set hashing function parameters for each PDMs
3. For each pattern in S with priority, starting with the
   first of the set:
   a. Generate indices using hash function for the
      PDM, taking two consecutive bytes at a time
   b. Map all the patterns into the PDMs:
      - The overlapping patterns must be mapped
        into correct PDMs according to their priority
      - If the entries for all the indices are not free,
        change the target PDM and go to step 3a
   c. If all PDMs are attempted, change the PDM
      hash parameters, reset memory, go to step 3
4. For each pattern in S without priority, starting with
   the longest pattern:
   a. Generate indices using hash function for the
      PDM, taking two consecutive bytes at a time
   b. Map all the patterns into the PDMs:
      - If the entries for all the indices are not free,
        change the target PDM and go to step 4a
   c. If all the PDMs are attempted, change the PDM
      hash parameters, reset memory, go to step 3
```

Fig. 11. A simple algorithm to map the preprocessed segments into PDMs

to yield cleaner indexing result. The system performance stays constant, regardless the size of the program.

Due to variety of possible algorithms and optimizations that can be applied to program the filter, we believe this step is also beyond the scope of this paper. Therefore, we present a few direct and effective algorithms used to map the entire pattern set defined in the Snort rules.

1) Pattern Preprocessing: For the above hardware, the long patterns must be broken into shorter segments of 17 bytes or less. Due to the priorities assigned to the PDM units, the short patterns do not have to be unique. However, eliminating duplicate patterns would save memory space. In order to identify each pattern with an unique index, the last segment of every pattern must be different.

We use the algorithm described in figure 10 to break the long patterns into smaller segments that fit in the PDMs. The algorithm produces a list of segments containing overlapping patterns. The overlapping patterns can assert detections in several PDMs in a single cycle. By assigning higher priority to the longer of any two overlapping patterns, the detection of the longer index also indicates the detection of shorter patterns. There is no need for priority for the non-overlapping patterns.

If any segment of the long pattern is an overlapping pattern, it must have the highest priority. Such priority is automatically assigned when the algorithm divides the pattern into maximum lengthed segments.

Once the list of pattern segments are generated, it can be used to generate index sequences for all the long patterns. When the long patterns are divided into smaller segments, the corresponding sequence of segment identifiers are recorded along with the time delay between subsequent segments and the type flags. These data are programmed into LPSM to keep track of the long patterns.

2) Programming the Filter: All the PDMs and the LPSMs are memory mapped. As far as the programmer is concerned, the filter can look like a large memory. The parameters of the hash functions can be also treated as a memory mapped location. Our implementation uses two ports to program the filter, one for the memory modules in the PDMs and LPSMs and the other for programming hash functions.

Before the filter is programmed, the data for the pattern matching modules must be mapped on to a virtual filter with same configuration. The mapping procedure is necessary to determine exact address locations for all data. Once the data is correctly mapped in to the virtual memory space, programming the filter is equivalent to writing into a memory.

The list of pattern segments, their length, and the control information from the preprocessing step are mapped on to the PDMs. Our mapper uses an algorithm in figure 11 to incrementally fill the PDM memory according to the pattern segment priority and the hash value. If the hash function fails to map the patterns, it simply changes the hashing parameters to re-map the patterns.

These simple algorithm mapped the entire Snort on to our initial implementation. However, the segments were not evenly distributed into all the memory modules. A better algorithm can use the distribution of the patterns in the memory and the frequency of possible indices for each pattern to efficiently

Module	Area	Units×Area	Cr-path
PDM Logic	0.075 mm^2	0.600 mm^2	<1.0 ns
LPSM Logic	0.024 mm^2	0.188 mm^2	<1.0 ns
PDM Mem	0.844 mm^2	6.752 mm^2	1.12 ns
LPSM Mem	0.168 mm^2	1.342 mm^2	1.12 ns
DPF Filter	-	8.882 mm^2	1.12 ns

TABLE I

ASIC DESIGN AREA OF THE FILTER USING $0.18\mu m$ TECHNOLOGY

Design	Device	BW (Gbps)	# of Bytes	Total Gates	Mem (kb)	Gates Byte
Cho-MSmith RDL+ROM	***Virtex 4 LX15***	***2.65***	***22340***	***4690***	***162***	***0.21***
Baker-Prasanna USC Unary	Virtex2 Pro100	1.79	8263	2892 ‡	0	> 0.35
Cho-MSmith ASIC SRAM	***ASIC***	***7.14 ****	***22340 †***	***11163***	***864***	***0.50***
Cho-MSmith Decoder	Spartan3 1500	2.00	20800	16930	0	0.81
Sourdis et al. Pred. CAM	Virtex2 3000	2.68	18031	19902	0	0.97
Clark-Schimmel RDL based	Virtex 1000	0.80	17537	19698	0	1.10
Franklin-Hutchings	VirtexE 2000	0.40	8003	20618	0	2.58
Gokhale et al. CAM	VirtexE 1000	2.18	640	~9722	24	15.19

* Bandwidth measured from ASIC design using 0.18μm library

† Patterns are using only about half of the maximum capacity of the filter

‡ Logic resource for the pattern index encoder is not accounted

TABLE II

PATTERN FILTER COMPARISON CHART [3], [7], [9]–[11], [14], [18]

map the pattern. Such mapping analysis will take longer execution time.

The sequences of indices and other control fields are mapped on to the LPSMs. Each index is mapped on to one LPSM pointing to one or more LPSMs that matches the corresponding next index. If there are patterns with same beginning indices, the programmer can choose to use only one LPSM to keep track of all the patterns until it branches off to different patterns. This optimization will allow the unused entries of the LPSMs to be used for other sequence of patterns.

After the data is successfully mapped on to the virtual filter, the memory values can be directly copied in to the filter memories.

C. Results

The hardware design is written in structural verilog and the programmer is written in C++. As described in the previous sections, the hardware is composed of 8 parallel units of PDMs and 8 parallel units of LPSMs.

As of June 2004, there are total of 1,729 unique patterns with lengths 2 to 122 bytes in Snort NIDS. The total number of bytes that the filter need to compare are 22,340 bytes. Using the simple algorithms, the programmer successfully configured the hardware to filter the entire set of patterns.

The pattern mapping would be more efficient if the memory usage distribution and the index information of patterns are used. Although our algorithm does not consider them in during the mapping process, the hardware still proved to be robust enough to store the rule set.

During the programming process, 374 long patterns are transformed into 752 short pattern segments; making the total number of patterns for the PDMs 2,107. Along with the segments, the LPSMs are programmed to make up to 7 state transitions.

The entire pattern set occupies approximately 50% of the PDM and 18% of LPSM memory, leaving enough space many additional patterns. In fact, given a complex algorithm that reuses duplicate substring, the filter can have more than double the number of patterns defined in the Snort NIDS.

On an AMD Athlon XP 1800+ processor under cygwin, the total runtime of the programmer to process and map the patterns into the virtual memory space is 771 msec. We generated the memory modules in Verilog with the contents of the virtual memory to verify our design in simulation environments.

We synthesized and routed the filter in ASIC using 0.18 μm technology in Cadence Synopsis tools. As shown in table I, the area and the critical path is limited to the memory modules in the design. The area for the processing modules account for less than 9% of the entire design while the rest of the area is occupied by the memory modules.

The critical path for the entire design is dictated by the memory which can run at a speed of 893 Mhz. Since the filter input can consume 1 byte of data at each cycle, the bandwidth of the filter is 7.144 Gbps.

Table II compares the FPGA resources needed for the filter against other recent pattern filters built using FPGAs. The new design is indicated as ASIC SRAM. With Snort NIDS patterns, our ASIC filter's gates per byte is relative comparable to the smallest design in FPGA. However, only half of the filter capacity is utilized with Snort NIDS. By applying new programming algorithm and adding new patterns to the set, the gates per byte may possibly decrease to below the smallest design.

V. PARALLEL STREAM SCANNER

In practice, the network traffic can be divided into seperate streams based on the packet information. For instance, according to study conducted by MCI in 1997, approximately 70% of the Internet traffic accounts for web transactions which uses port 80 under TCP/IP [19]. Using the static header information, two independent data streams can be formed by seperating all the web traffic from the rest. Accordingly, the rule set can be divided into two sets for use in two parallel filters to scan each stream. Therefore, under normal network traffic condition, one can expect bandwidth to increase by 43%. If the web traffic accounts for 50% the same hardware would be able to filter at two times the rate.

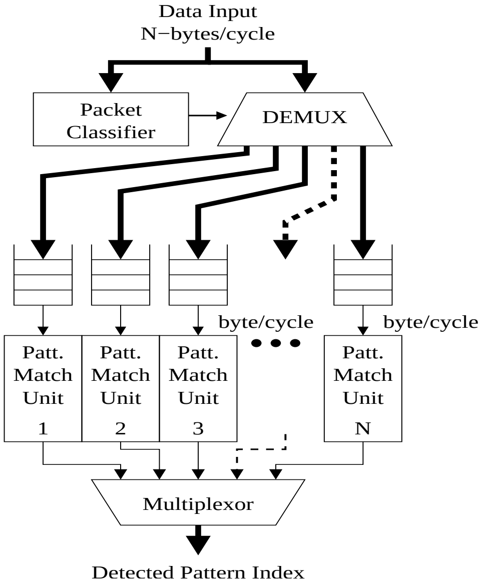


Fig. 12. Parallel Stream Scanner is composed of packet classifier followed by parallel pattern matching units.

A. Packet Classifier

If the network traffic statistic is available, we can structure our filters based on the information to increase the performance. As the Internet example suggests, our approach divides the network traffic as well as the pattern matching unit according to the traffic statistics to effectively increase the performance by scanning several independent streams in parallel.

Figure 12 shows the architecture that parallelizes the pattern detection process in terms of streams. The packet classifier examines the static information of the packet to de-multiplex the packets to appropriate pattern match engine. Given N-parallel pattern matching units, the packet classifier must de-multiplex the data at N times the filtering rate of individual pattern matching unit. Since the classifier does not have to classify the packet for every byte, its bandwidth can be multiplied by widening the input data bus.

B. Pattern Matching Units

Each output of the de-multiplexor will be at a higher bandwidth than the individual pattern matching unit can filter. However, if we assume the traffic statistic holds true, the traffic load should be divided equally among all the pattern matching units. Therefore, we can reduce the output bandwidth of the de-multiplexor to match the maximum filtering rate of each pattern matching unit by using a small FIFO.

As network packets are classified according to the statistics, the rule set need to be assigned to different pattern matching unit. Then, each pattern matching unit can be smaller than the original engine because of the smaller rule set. By analyzing the target network and the rule set, the sizes for each unit can be determined to optimize the space and performance.

C. Summary

Table I shows that the memory accounts for 91% of the total die area in our initial implementation. Since memory in each pattern matching unit will be adjusted according to the rule set size, the entire design size will be increased due to redundant hashing logic, packet classifier, and the additional FIFOs.

When the network traffic load is balanced equaly over all the matching units, the maximum bandwidth of the entire filter is multiplied by the number of the units. On the other hand, when the traffic is not distributed evenly, only one unit may be doing all the work, thus making the minimum bandwidth to equal the filtering rate of one unit. For example, if our initial pattern matching unit was successfully divided into 4 smaller units, the maximum bandwidth of the system would be 28.576 Gbps whereas the minimum bandwidth will equal to the initial implementation of 7.144 Gbps.

VI. Conclusion

In this paper we describe a novel architecture for pattern matching co-processor for network intrusion detection system. The co-processor is RAM-based design that is programmable using the list of substrings and the state transistions. Its efficient pattern matching engine is capable of filtering the multiple gigabit network traffic. Since the patterns are programmed by changing the contents of the RAM, the architecture can be used to implement designs in FPGA as well as ASIC.

We have shown that our pattern filter is capable of yielding performance that surpasses the most recent FPGA implementations while enabling the users to program it without having to regenrate and reconfigure the hardware. Such quick configuration may become critical, as the rate of emergence of new attack increase.

We further developed our architecture by presenting a simple structural modification to the initial design to obtain higher bandwidth. By effectively dividing the network traffic based on the network statistic, we show that the maximum bandwidth of the new design can be multiplied by the number of smaller parallel units.

References

[1] G. Memik, S. O. Memik, and W. H. Mangione-Smith, “Design and Analysis of a Layer Seven Network Processor Accelerator Using Reconfigurable Logic,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa Valley, CA: IEEE, April 2002.

[2] M. Roesch, “Snort - Lightweight Intrusion Detection for Networks,” in *USENIX LISA 1999 conference*. http://www.snort.org/: USENIX, November 1999.

[3] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, “Specialized Hardware for Deep Network Packet Filtering,” in *12th Conference on Field Programmable Logic and Applications*. Montpellier, France: Springer-Verlag, September 2002, pp. 452–461.

[4] Y. H. Cho and W. H. Mangione-Smith, “Deep Packet Filter with Dedicated Logic and Read Only Memories,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa Valley, CA: IEEE, April 2004.

[5] D. Watson, M. Smart, G. R. Malan, and F. Jahanian, “Protocol Scrubbing: Network Security through Transparent Flow Modification,” in *IEEE/ACM Transactions on Networking*. ACM Press, April 2004.

[6] R. Sidhu and V. K. Prasanna, “Fast Regular Expression Matching using FPGAs,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa Valley, CA: IEEE, April 2001.

[7] R. Franklin, D. Carver, and B. L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*. Napa Valley, CA: IEEE, April 2002.

[8] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System," in *13th Conference on Field Programmable Logic and Applications*. Lisbon, Portugal: Springer-Verlag, September 2003.

[9] C. R. Clark and D. E. Schimmel, "Scalable Parallel Pattern-Matching on High-Speed Networks," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa Valley, CA: IEEE, April 2004.

[10] Z. K. Baker and V. K. Prasanna, "A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa Valley, CA: IEEE, April 2004.

[11] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," in *12th Conference on Field Programmable Logic and Applications*. Montpellier, France: Springer-Verlag, September 2002, pp. 404–413.

[12] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," in *IEEE Hot Interconnects 12*. Stanford, CA: IEEE Computer Society Press, August 2003.

[13] J. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks, "Internet Worm and Virus Protection in Dynamically Reconfigurable Hardware," in *Military and Aerospace Programmable Logic Device (MAPLD)*. Washington DC: NASA Office of Logic Design, September 2003.

[14] Y. H. Cho and W. H. Mangione-Smith, "Programmable Hardware for Deep Packet Filtering on a Large Signature Set," in *First IBM Watson P=ac2 Conference*. Yorktown, NY: IBM, October 2004.

[15] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," in *Communications of the ACM*. ACM Press, June 1975, pp. 333–340.

[16] N. Desi, "Increasing Performance in High Speed NIDS: A look at Snort's Internals," Feb 2002.

[17] J. Moscola, J. Lockwood, R. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa Valley, CA: IEEE, April 2003.

[18] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa Valley, CA: IEEE, April 2004.

[19] K. Thompson, G. Miller, and R. Wilder, "Wide-Area Internet Traffic Patterns and Characteristics," *IEEE/ACM Transactions on Networking*, pp. 10–23, November 1997.

Young H. Cho Young cho attended the University of California, Berkeley, where he received B.A. in Computer Science in 1996. While he was in Berkeley, he was part of research group called Networks of Workstations. From 1996 to 1999, he was employeed by a start-up network company, Myricom, Inc., where he developed several commercial and research projects involving high-performance network and computers. At the end of 1999, he returned to academia to receive M.S.E. in Computer Engineering from the University of Texas, Austin. Since 2001, he has been working on his Ph.D. in Electrical Engineering from the University of California, Los Angeles. His research is in high-performance network security and embedded computing systems.

William H. Mangione-Smith William H. Mangione-Smith attended the University of Michigan, Ann Arbor, where he received a B.S.E. degree in electrical engineering in 1987 and the M.S.E. and Ph.D. degrees in computer science and engineering in 1991. From 1991 to 1995, he was employed by Motorola, Inc., where he participated in the design of the Envoy Personal Digital Assistant. Since 1995, he has been a professor in the Electrical Engineering Department at the University of California, Los Angeles.

Prof. Mangione-Smith has been honored with the 1998 NSF CAREER Award in support of his research program, ”PSICs: Problem-Specific Integrated Circuits”. He is a member of the IEEE and the ACM.