

Fast Reconfiguring Deep Packet Filter for 1+ Gigabit Network

Young H. Cho and William H. Mangione-Smith

{young,billms}@ee.ucla.edu

University of California, Los Angeles

Department of Electrical Engineering

Los Angeles, California 90095

Abstract

Due to increasing number of network worms and virus, many computer network users are vulnerable to attacks. Unless network security systems use more advanced methods of content filtering such as deep packet inspection, the problem will get worse. However, searching for patterns at multiple offsets in entire content of network packet requires more processing power than most general purpose processor can provide. Thus, researchers have developed high performance parallel deep packet filters for reconfigurable devices. Although some reconfigurable systems can be generated automatically from pattern database, obtaining high performance result from each subsequent reconfiguration can be a time consuming process. We present a novel architecture for programmable parallel pattern matching co-processor. By combining a scalable co-processor with the compact reconfigurable filter, we produce a hybrid system that is able to update the rules immediate during the time the new filter is being compiled. We mapped our hybrid filter for the latest Snort rule set on January 13, 2005, containing 2,044 unique patterns byte make up 32,384 bytes, onto a single Xilinx Virtex 4LX - XC4VLX15 FPGA with a filtering rate of 2 Gbps.

1 Introduction

An effective security measure for current network attack is deep packet filtering [16]. Deep packet filters are design to examine not only the headers but also the payload of the network packets to detect the patterns defined in the signature set. For example, Snort network intrusion detection system (NIDS) [18, 5, 3], can be configured to detect several worms undetectable by traditional firewalls.

The Internet traffic is made of streams of fragmented packets consisting header and payload. Since attack can

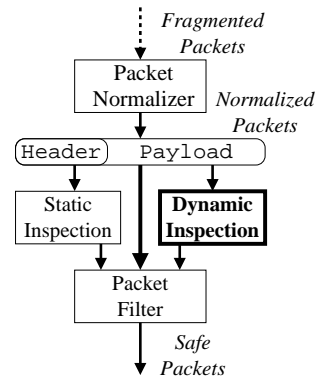


Figure 1. Deep Packet Inspection

span more than one packet of a stream, every stream needs to be reassembled before applying the deep packet inspection. There some class of attacks that use unconventional protocol features and packet fragmentation to avoid the intrusion detection system. One such attack uses overlapping fragmented IP packets. Such attacks can be eliminated by normalizing the packets. Packet normalization produces consistently clean network traffic without abnormalities [24].

Figure 1 shows an effective deep packet inspection steps which begins with packet normalization followed by static and dynamic inspections. Static inspection step classifies the incoming packets using the header information, while dynamic inspection searches through the payload of the packet to find the patterns defined in the attack signature database [20]. Analysis of the detection results from both inspection process will lead to quicker and effective intrusion detection.

Most available deep packet inspection systems use one or more general purpose processors to deploy signature-based filtering software. Although this software can be configured to detect several network attacks, the most processors are not powerful enough to sustain acceptable filtering

*This research work was supported by NSF Grant #CCR-0220100.

rate for 1+ gigabit per second network. For instance, Snort NIDS with 500 patterns can sustain a bandwidth less than 50 Mbps using a dual 1 GHZ Pentium 3 system [5].

With emergence of new worms and virus, the rule set is constantly updated. Therefore, deep packet filter systems need an ability to be re-programmed. Since software implementation on general purpose processors cannot meet the performance requirement of the application, field programmable gate array (FPGA) is an excellent platform for building specialized parallel processing system that can be reconfigured [15].

However, unlike the software system, the process of re-compiling the updated FPGA design can be lengthy. For recently developed reconfigurable filters, adding or subtracting any number of rules requires recompilation of the entire design. Given demanding performance constraints, the compilation process can take more than several hours to complete. Such latency in compilation time maybe acceptable for most network today. As new attacks are released at a higher frequency, it may be necessary for the system to update its pattern database much faster.

The contribution of the paper is a novel architecture of programmable pattern matching co-processor. Based on the architecture, we implement a scalable co-processor best suited for FPGA. We combine this processor with our area efficient reconfigurable pattern matcher [4] to form a hybrid system. When new patterns are added to the set, they can be immediately updated and detected by the co-processor. This update process gives the compiler enough time to generate a new filter that meets specified area and performance constraints.

We begin our discussion in section 2 with a brief overview of related works in reconfigurable pattern matcher; including our area efficient design. In section 3, we describe our novel architecture of pattern matching co-processor. Then in section 4, we present the implementation of our hybrid system that combines the reconfigurable implementation with the co-processor. We discuss area and performance results as well as algorithms used to immediately update the detectable pattern set. Then we conclude in section 5 with a comparative chart for several other related work.

2 Related Work

Snort is one of the most widely used open source deep packet filtering software today. Therefore, almost all researchers have used it as a basis for building high performance pattern matchers on FPGAs.

2.1 Header Classifiers

Although our work is focused on accelerating the dynamic inspection system, it is important to mention that the static inspection of the network traffic is crucial in deep packet inspection.

The header classifier is utilized for many network applications. For the purpose of NIDS application, there are several commercial products such as PMC Sierra ClassiPI [12] and Broadcom Strada Switch II[11]. Although they are capable of limited dynamic inspection, they are best suited for static inspection.

For the purpose of NIDS, there have been a few efforts to accelerate the header classification process on FPGAs. The most popular method uses Ternary Content Addressable Memory (TCAM) to classify packets. There are several research projects that utilizes TCAM in different variations of architecture and algorithm to classify packets at a higher rate [13, 23, 20].

Without an efficient header classification, dynamic inspection engines may process the packet payload unnecessarily. By effectively utilizing the header classifier in conjunction with the payload pattern matcher, more efficient deep packet inspection system can be built.

2.2 Reconfigurable Pattern Matchers

Recent research results have shown that the patterns can be translated into non-deterministic and deterministic finite state automata to effectively map on to FPGAs to perform high speed pattern detection [19, 15, 9].

Researchers have also shown that the area efficient pattern detectors can be built by optimizing the group of byte comparators that are pipelined according the patterns [5, 21, 22, 2, 6].

Few other researchers have implemented filters in FPGA that use built-in memories to detect patterns. Gokhal et al. implemented a re-programmable pattern search system using content addressable memories (CAM) [10]. Dharmapurikar et al. built a pattern filter named Bloom filter using specialize hash functions and memories [8, 14, 17].

Our earlier work uses the chains of byte comparators and read-only-memory (ROM) to reduce the amount of logic by storing parts of the data in the memory [3, 4].

Many of the above implementations performs over gigabit per second filtering rate. However, unless the implementations does not require reconfiguration [10, 8], the FPGA design re-compilation can be a very lengthy process, especially if the time and area constraints are very demanding.

2.3 Reconfigurable ROM based Filter

Our compact filter design [4] (Figure 2) consists of chain of pipelined prefix comparators followed by address gen-

erator, suffix ROM, and the suffix comparator. The initial portions of data which we referred to as prefix is detected by the prefix comparators. Then the address for the suffix ROM is generated according to the match. The rest of the data is compared with the suffix in the suffix comparator for an exact match.

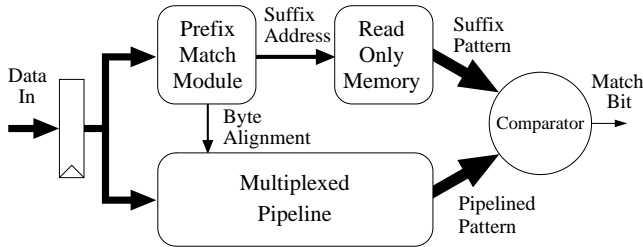


Figure 2. Block diagram of Read-Only-Memory based inspection unit

Since many FPGAs, today, contain built-in block memory modules, this design utilizes the FPGA resources more efficiently than the purely logic based designs, allowing our filters to map into smaller and less expensive devices.

We generated the filter for 2,851 rules which consists of 2,044 unique Snort patterns composed of 32,384 bytes. It mapped, place, and routed into a single Xilinx Virtex 4 LX - XC4VLX15 FPGA. The design used 10 block memories, 6,877 four input look-up-tables (LUT4), and 7,882 flip-flops. Due to its fine grained pipelines, its sustainable maximum filtering rate is 2.31 Gbps.

In addition to these measurements, the total compilation time of the design is pertinent to this paper. The Xilinx ISE v6.3 tool took total of 30 minutes to compile the design on a AMD Athlon 1800+XP machine with 1 GB of PC2700 SDRAM.

3 Pattern Matching Co-Processor

In this section, we present a compact and programmable pattern search co-processor architecture for FPGA as well as ASIC. Unlike the reconfigurable logic solutions described in the previous sections, the filters using this architecture does not need logic reconfiguration to make pattern set updates. Its pattern database is programmed and updated by changing its memory content.

3.1 Pattern Detection Module

The basic pattern detection module (PDM) is shown in figure 3. The function of the pattern detection module is to efficiently detect specified pattern segments using programmable hash functions and memory, followed by a discrete string comparator.

At every clock cycle, part of the input data is hashed to generate an index. The index is a pointer to an entry in the memory where the corresponding pattern segment is stored. The retrieved segment from the memory is compared with the input data. When there is a match, the index is forwarded as an output to indicate an identity of the segment.

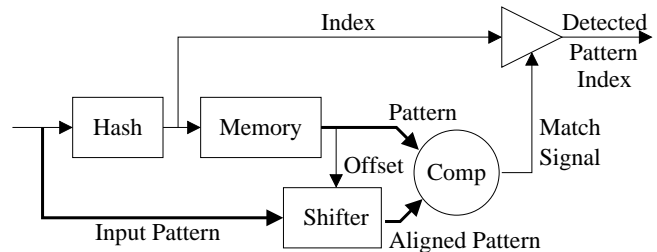


Figure 3. Pattern Detection Module

The architecture is parametrized and cascaded so that the length of the patterns does not have to be fixed. The maximum length of the input bytes that is used to generate the hashed index determines 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 bytes of the input pattern are hashed to generate an index, the PDM can be configured to detect maximum of 65,536 patterns with the minimum length of two bytes.

3.1.1 Hashed Index

The hash function can be implemented to always generate an index using substring at a fixed byte offset of input patterns. Such hashing method will allow only one unique index to be generated for any given pattern. This would limit the number of patterns that can be stored on the same PDM.

In practice, we can expect that each pattern is made of set of smaller unique substrings. As shown in figure 4, there will be more choices for hash index when the hash function is allowed to use any substring in the set. Given several possible hash indices for each pattern, the system can apply statistical analysis to the patterns to map them on the PDMs more efficiently.

Since the indices are generated using the bytes from different offsets, the timing of the identification output may not be synchronized to the pattern location. By using the offset value with the switched pipeline as the one shown in figure 5, the index output timing can be adjusted to indicate the correct order of detection.

3.1.2 Prioritized Parallel Modules

As the number of patterns increases, some may not be mapped on to the same PDM due to limited number of

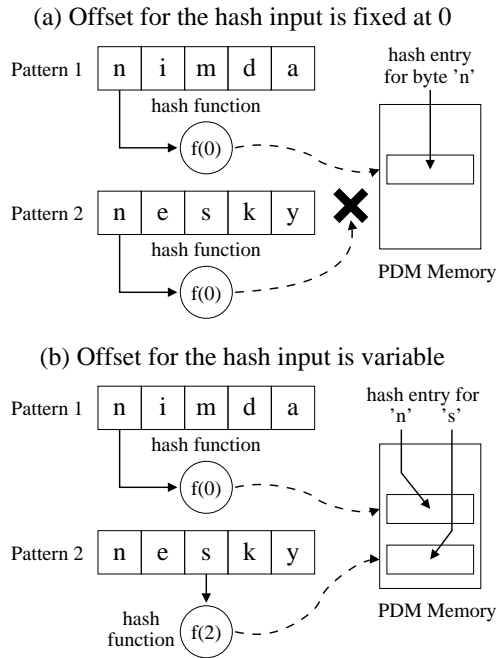


Figure 4. (a) The offset for the hash input is fixed at 0. (b) The offset is variable, thus first pattern can be hashed at 0 and the second pattern can be at 2.

unique substrings combinations. Therefore we may need more than one PDM to detect patterns in parallel. Despite the same hash index, only one PDM will signal a match since no two patterns should be the same.

However, for some patterns, more than one PDM can produce valid indices during the same cycle. This is only true when one pattern matches the beginning substring of another pattern; in another words, the long pattern “overlaps” the short pattern from the starting byte. We refer to such patterns as “overlapping patterns.” Therefore, when more than one index is detected, it is sufficient to output the index for the longest pattern.

As shown in figure 6, multiplexers are cascaded to implement priority. 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.

Each PDM unit is capable of detecting patterns of lengths that are less than or equal to that of the widest memory module of all the PDMs. Given a typical set of patterns, designer may choose to use different sized memory for the PDMs. By statistically analyzing the patterns, the logic resource may be used more efficiently.

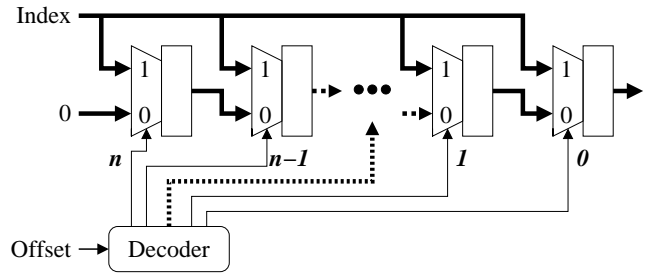


Figure 5. Switched Pipeline

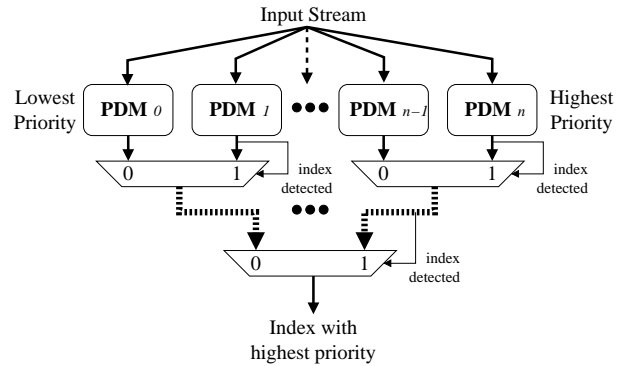


Figure 6. Parallel PDMs with priority

3.2 Long Pattern State Machine

If the lengths of the patterns vary building PDMs using the memory wide enough to store the longest pattern would result in a poor memory and logic utilization. In this section, we describe a cascaded module called long pattern state machine (LPSM) that is used to detect patterns that are longer than the width of PDM memories.

Since not all detected segments are part of a long pattern, the segment indices can be hashed to increase LPSM memory utility. The LPSM examines the sequence of segment indices for the correct ordering and the timing to detect the corresponding long pattern. There are two ways to detect index ordering, predictive and retrospective.

In LPSM, each state corresponds to an entry in its memory. The memory entries in predictive LPSM contains information on its next state while retrospective LPSM contains information on its previous state. 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.

3.2.1 Keyword Tree

Before we can detect the order of indices, the long patterns need to be divided into several short pattern segments. If we keep track of the order and the timing of the segment sequence, we can effectively detect the corresponding long

pattern.

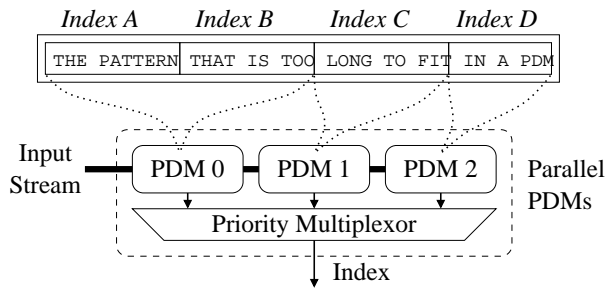


Figure 7. Divided segments of the long pattern maybe detected by different PDMs

As in figure 7, the long pattern is divided into smaller segments that fit in to a specific PDM. These segments are stored in the PDMs along with flag bits that indicate that they are segments of long patterns.

One efficient way to divide and represent the patterns is Aho and Corasick's keyword tree [1]. A keyword tree is used in many software pattern search algorithms, including the Snort IDS [7]. This algorithm is already used in our reconfigurable implementation to reduce the logic area [3].

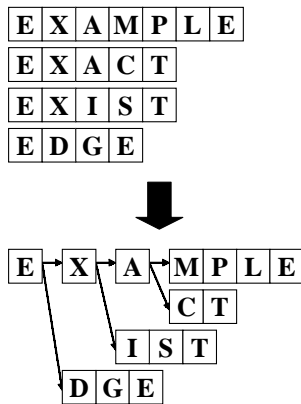


Figure 8. An example of Aho and Corasick's Keyword Tree: 6 bytes are optimized away.

A keyword tree in figure 8 shows how it can optimize the memory utility by reusing the 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. We apply the key concept of keyword tree to build the set of pattern segments from the long patterns that fits in the PDM memories.

3.2.2 Predictive LPSM

Predictive LPSM determines the order of indices sequence using an intuitive algorithm. As shown in figure 9, the expected next index is forwarded to the switched pipeline like the one used in PDM to add the predicted delay. When the next index reaches the end of the pipeline, it is compared with the current index for a match.

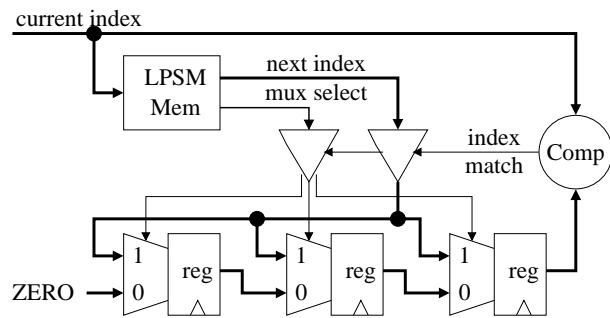


Figure 9. Predictive LPSM

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

More than one LPSM can run in parallel to detect more than one sequence of indices. The match bit is sent across the parallel LPSMs to the ones known to contain the predicted next index. When any of the LPSM receives the match bit, its expected next index is forwarded to the pipeline regardless of the result in its own comparator.

Such structure makes parallel predictive LPSM a natural platform to map regular expression. Regular expression can be represented in the form of NFA [19, 9]. All the inputs to the NFA can be recognized by the PDM as sequence of short segment while the transition can be mapped on the parallel LPSMs. For the each index entry, each LPSM can point to the next index that is the next node of the NFA. In similar fashion, DFAs can also be mapped in to the module [17].

For instance, 10 shows the node with edges that points to it self and to another node. Such finite automata can be represented in the parallel LPSM, where an entry on one unit points to itself and the same entry on another unit points to the next index.

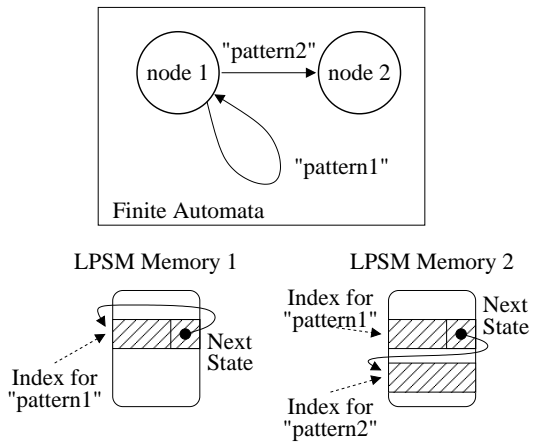


Figure 10. Regular expression: (“pattern1”)+ “pattern2” - one or more instance of “pattern1” followed by “pattern2”

3.2.3 Retrospective LPSM

Although retrospective LPSM may not be an intuitive choice for mapping finite automata with cyclic paths, it is a preferable module for a pattern keyword tree, especially if nodes of the tree consist of many children nodes. If all keywords of a given tree has less children than the number of parallel LPSMs, predictive LPSM may be sufficient; otherwise, number of parallel predictive LPSMs must be increased.

In retrospective LPSM, the keyword tree is mapped on to the LPSM memory in a bottom-up fashion. Therefore, as long as all the indices are addressable in the LPSM, the keyword tree can be successfully mapped.

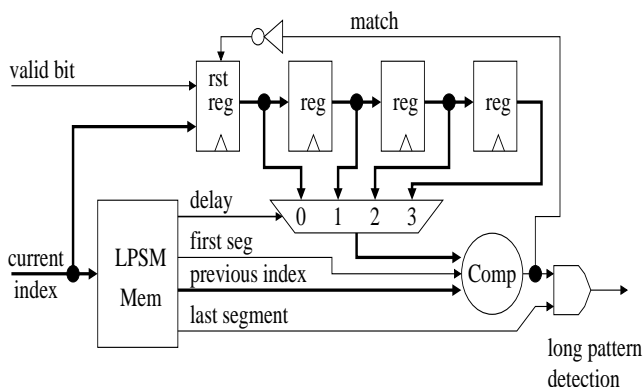


Figure 11. Retrospective LPSM

As shown in figure 11, the retrospective LPSM, first, forwards the previously detected index according to the delay information stored for the current index. If the previous in-

dex is valid at that stage of the pipeline, it compares the index value with the expected index stored in the memory. When there is a match, valid bit for the current index is passed to the next stage of the pipeline. Otherwise, the valid bit and the detected current index is invalidated.

The first segment bit causes the comparator to always output a match. By asserting the first segment bit of the first index entry, the order verification process is initiated.

3.3 System Integration

Figure 12 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. The delay is added to the PDMs so that the timing of the short pattern segment detection is the same as the long pattern, so that the output may be shared.

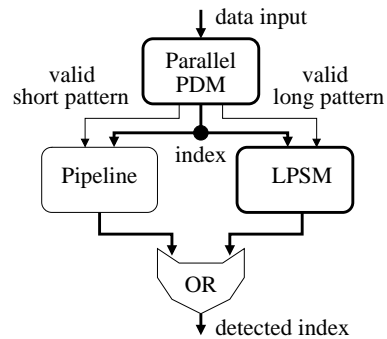


Figure 12. Block Diagram of Short and Long Pattern Filter

Unlike the reconfigurable logic designs, which requires recompilation of the design file, the patterns can be updated by changing the content of the memories in LPSMs and PDMs. Therefore, the above system takes much less time to update the inspection rule set than the reconfigurable solution.

4 Hybrid system

Snort is one of the most widely used network intrusion detection system (NIDS) that uses deep packet inspection. On January 13, 2005, the Snort rule set contained 2,044 unique string patterns consisting 32,384 bytes. We implement the entire pattern database by combining the reconfigurable ROM based filter with the co-processor. This implementation takes advantage of the area efficiency in the reconfigurable ROM based filter and the programmability co-processor to allow fast rule updates.

All the patterns at the time of recompilation should be translated into reconfigurable ROM based filter to minimize the area. In order to quickly update the system after the FPGA reconfiguration, a small pattern matching co-processor can work independently from the ROM based filter. Since two modules are independent, they can reside in the same or separate devices. Since the co-processor does not require reconfiguration, it can be an ASIC core either on a separate chip or embedded in an FPGA. For our purpose, we implement both modules on the same FPGA. Therefore, the co-processor design considers the primitive components of the target FPGA to produce a compact design.

Signature Set	Rules	Unique Patt	Tot. Bytes
Oct 19,04	2,707	2,031	32,168
Jan 13,05	2,851	2,044	32,384
Oct 04-Jan 05	-	17	91
Jan 05-Oct 04	+	30	307

Table 1. Between Oct 19, 2004 and January 13, 2005, 17 unique patterns were taken out, while 30 new patterns were added in to the database

We collected two Snort rule set for different period. The pattern sets are extracted from January 13, 2005 and October 19, 2004 versions of rule set. We compared the sets and produced one list containing all the patterns that were taken out and the other containing all the new patterns added since October. As indicated on the table 1, 17 old patterns were discarded while 30 new ones were added to the set.

Given the above pattern sets, we demonstrate our design by carrying out the recompilation process back in October 19, 2004. Therefore, our system consists of ROM based filter for October and a co-processor which is used to hold all the additional signature until January 13, 2005.

4.1 ROM based filter

Our previous papers [3, 4] has detailed descriptions of our ROM based filter design. Therefore, we only discuss its parameters and resource usage for the module.

The size of the primitive block memory unit of the Xilinx Virtex 4 FPGA is 18 kilobits. It can be configured to several different widths and depths. For memory unit with 256 entries, each block can be configured to have width of 9 bytes. By concatenating the memory blocks, the width of the memory can be efficiently extended.

Currently, we do not automate the partitioning process which can differ based on the FPGA resources. Finding an optimal configuration for the pattern set is beyond the scope of this paper. Thus, we experimented with several configurations to choose the configuration with the best memory

utilization. We instantiate ten blocks of memory to map four partitions of the pattern set to obtain 95% memory utilization.

4.2 Pattern matching co-processor

There are two design issues with co-processor, the hardware configuration and the software mapping algorithm.

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 also beyond the scope of this paper. Therefore, we attempt to describe one system we have implemented to demonstrate the effectiveness of our system for Snort NIDS.

4.2.1 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 equal to optimally use the fixed size primitive block memories of the FPGA.

Considering the all dimensions of block memories, we choose the dimension of the memory in each PDM to be 9 bytes by 256 entries. Since the address pin for each memory is 8 bits, our hash function uses the input byte as its output. Therefore, the minimum length of the pattern detectable with our filter is one byte.

If the target pattern set do not have uniform distribution of bytes in the pattern, hash function can generate an index by using more than one byte. Using such hash function will increase the memory utilization by introducing more diversity in the index. However, the minimum length of the detectable pattern must be greater or equal to the hash function input.

Nine bytes of each entry need to be partitioned to hold not only the patterns but its type, length, and hash function input offset. By assigning 2 bits for type information, and

3 bits each for the length and offset, maximum length of detectable pattern is 8 bytes.

4.2.2 LPSM parameters

Since our application does not have any cyclic regular expressions, we use retrospective LPSM to detect long patterns. We use single LPSM with a dimension of 18 bits by 1024 entries. All addresses from four PDMs are mappable with such configuration. Therefore, the indices are not hashed and forwarded as an address to an LPSM entry.

16 of 18 bits of each memory entry is used to store the current segment type, the previous segment index, the delay between the previous and the current segment, and memory entry valid flag.

4.2.3 Pattern segmentation

Once the hardware parameters are determined, the resulting data-path 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 to yield cleaner indexing result. The system performance stays constant, regardless the size of the program.

For the above hardware, the long patterns must be broken into shorter segments of 8 bytes or less. Because of 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 a unique index, the last segment of every pattern must be different.

For simplicity, we use a heuristics to build a keyword tree. There are three of things to consider when long patterns are segmented into short patterns. Firstly, the last segment of every the long pattern must not overlap any other segments. By processing the patterns such way, the filter is guarantee to only detect single long pattern. Then, we should attempt to segment the patterns to have the maximum length. With longer patterns, the PDMs have more choices for hashed index for a given pattern. Lastly, the segments in the middle of one long pattern should not be used as a middle segment of another long pattern. Since there is only one entry for one index, such patterns cannot be mapped into the same LPSM unit. With these considerations, an algorithm can divide the long patterns in to several short patterns that fit in the PDMs.

First, we scan the last segment of maximum length to build the list of keywords. By iteratively comparing the list with the segment, we can check and build a list of unique keywords in a single pass of the patterns. When there is an overlapping segments, then we attempt to modify it

by shortening the segment by one byte until the minimum length is reached.

Once all the last segments are defined, the rest of the segments can be added to the list. We segment the patterns so that, all but the first segment, are not allowed to overlap any of the other previously defined segments. When there is an overlap, we attempt to change the segmentation by moving the moving the segment alignment forward or by reducing the segment size from the start or the end of the segment.

As the list of pattern segments are generated, index sequences along with all the necessary information for retrospective LPSM is recorded for every long pattern. In order to the store the pattern segments and index sequences to the memory, a mapping algorithm must be used to fit the segments into the available PDM entries.

4.2.4 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.

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 incrementally fill the PDM memory according to the pattern segment priority and the hashed index.

1. Produce an histogram vector (A) of all the bytes in the entire pattern set
2. For each pattern, produce an histogram vector (B) of all the bytes in the pattern
3. Multiply each index of vector (A) with (B) to produce vector (C)
4. Assign the index with the smallest non-zero value in (C) as the hashed index for the segment
5. Produce a vector (D) indicating the number of segments hashed to each index
6. Find all the indices that have more segments than the maximum number of PDMs
7. For the indices in 6, attempt to rehash any of the segments into indices with less segments until the number of segments equal the maximum allowed

Figure 13. Pattern segmentation heuristics

While we map the hashed indices into the PDMs, we must consider that there are four parallel units. If there are more than segments assigned to the index, all of them cannot be mapped on to the co-processor. Therefore, we apply an intuitive heuristic described in figure 13 to find the index distribution that fits in the co-processor.

4.3 Results

The hardware design is automatically produced in structural VHDL. The pattern mapper is written in C++. As described in the previous sections, the hardware is composed of 4 parallel units of PDMs connected to a single unit of retrospective LPSM.

In October 19, 2004, there were total of 2,031 unique patterns with lengths 1 to 122 bytes in Snort NIDS. The total number of bytes that the filter need to compare were 32,168 bytes. By January 13, 2005, additional 30 patterns consisting 307 bytes were added to the database while 17 patterns were deleted.

To demonstrate our system, the reconfigurable ROM based filter for October of 2004 rule set is generated. A single unit of pattern matching co-processor designed is compiled with the ROM based filter on to the same FPGA. Then the co-processor is programmed to detect the 30 new patterns from January 13, 2005. The entire pattern set occupies only 4.9% of PDM and 3.1% of LPSM memory, leaving enough space for many additional patterns. However, to more effectively use the FPGA resource, the system engineer may choose to automatically regenerate the structure VHDL code using the latest rule set, and recompile with required constraints.

Design	Device	ROM (Bytes)	Co-proc (Bytes)	HW comp Time	SW map Time
Rom+Coproc (January 05)	Virtex 4 LX15	32168	307	28 min	< 10ms
Rom+Coproc (January 05)	Virtex 4 LX25	32384	0	26 min	0

Table 2. Compilation and mapping execution time for two versions of the system

As shown in table 2, we measured the compilation times October 2004 and January 2005 versions of our system. The machine we used for hardware compilation and software mapping is an AMD Athlon XP 1800+ machine with 1 GB PC2700 SDRAM. Both compilation took total of 30 minutes to complete in Windows version of Xilinx ISE v6.3. Under cygwin, the total runtime of the co-processor mapping software for the new patterns measured to 0 ms. Since 10 ms is the lowest time resolution measurable through C++

function, we can conclude that the execution time is less than 10 ms.

Table 3 compares the FPGA resources needed for the filter against other recent pattern filters built using FPGAs. The timing constraints placed on the designs is 4 ns. All our designs met our timing constraints, allowing us to clock the design at 250 MHz. To show the effectiveness of our co-processor, we also compiled our older co-processor design using 8 parallel PDMs and 8 predicative parallel LPSM. This system was intended for ASIC process. Due to sparse pipelining and less efficient overall design, the filtering rate is half the newer designs. However, as indicated in the table we can map the entire pattern database from June 2004 using only half of the memory available on to the system.

Design	Device	BW (Gbps)	# of Bytes	Total Gates	Mem (kb)	Gates Byte
<i>Cho-MSmith Rom (Oct 04)</i>	<i>Virtex 4 LX15</i>	2.17	32168	6727	184	0.21
<i>Cho-MSmith Rom (Jan 05)</i>	<i>Virtex 4 LX15</i>	2.17	32384	6877	184	0.21
<i>Cho-MSmith Rom + Coproc (Oct 04 + update)</i>	<i>Virtex 4 LX15</i>	2.08	32384 *	8480	276	0.26
<i>Cho-MSmith Rom+Coproc (Jan 05)</i>	<i>Virtex 4 LX25</i>	2.08	32384 ††	8637	276	0.27
Baker-Prasanna USC Unary	Virtex2 Pro100	1.79	8263	2892 ‡	0	> 0.35
<i>Cho-MSmith Coproc (Jun 04)</i>	<i>Virtex 4 FX20</i>	1.00	22340 †	11653	885	0.52
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

* Patterns are using less than 5% of the maximum capacity of filter
† Patterns are using about half of the maximum capacity of the filter
†† No patterns are occupying the memories
‡ Logic resource for the pattern index encoder is not accounted

Table 3. Pattern filter comparison chart [9, 10, 5, 2, 4, 22, 6]

5 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 can be programmed using the list of substrings and the sequence of index transitions. 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.

However, area efficiency in FPGA is best achieved by using the reconfigurable ROM based design that we have presented in earlier publications [3, 4]. Therefore, we propose to combine the ROM based filter with a smaller co-processor to realize fast pattern database updates. While the co-processor is used to detect the new patterns, the system engineers can spend longer time to generate and compile the new system for reconfiguration.

We have shown that our pattern filter is capable of yielding performance that surpasses the most recent FPGA implementations while enabling the users to update the system immediately. Such short configuration time may soon become necessary, as the rate of emergence of new attack increase.

References

- [1] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. In *Communications of the ACM*, pages 333–340. ACM Press, June 1975.
- [2] 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, April 2004. IEEE.
- [3] 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, April 2004. IEEE.
- [4] 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, October 2004. IBM.
- [5] 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*, pages 452–461, Montpellier, France, September 2002. Springer-Verlag.
- [6] 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, April 2004. IEEE.
- [7] N. Desi. Increasing Performance in High Speed NIDS: A look at Snort's Internals. In <http://www.snort.org>, Feb 2002.
- [8] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *IEEE Hot Interconnects 12*, Stanford, CA, August 2003. IEEE Computer Society Press.
- [9] R. Franklin, D. Carver, and B. L. Hutchings. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *IEEE Symposium on Field-programmable Custom Computing Machines*, Napa Valley, CA, April 2002. IEEE.
- [10] 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*, pages 404–413, Montpellier, France, September 2002. Springer-Verlag.
- [11] B. Inc. Strada Switch II BCM5616 Datasheet. In <http://www.broadcom.com>. Broadcom Inc., 2001.
- [12] P. S. Inc. Preliminary PM2329 ClassiPi Wire-speed Performance Application Note. In <http://www.pmc-sierra.com>. PMC Sierra Inc., 2001.
- [13] H. Liu. Efficient Mapping of Range Classifier into Ternary-CAM. In *IEEE Symposium on High Performance Interconnects*, Stanford, CA, August 2002. IEEE.
- [14] 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, September 2003. NASA Office of Logic Design.
- [15] J. W. Lockwood. Evolvable Internet Hardware Platforms. In *Proceedings of the 3rd NASA/DoD Workshop on Evolvable Hardware*, pages 271–297, Long Beach, CA, 2001. Department of Defense.
- [16] 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, April 2002. IEEE.
- [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, April 2003. IEEE.
- [18] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX LISA 1999 conference*, <http://www.snort.org/>, November 1999. USENIX.
- [19] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2001. IEEE.
- [20] H. Song and J. W. Lockwood. Efficient Packet Classification for Network Intrusion Detection using FPGA. In *ACM International Symposium on FPGAs*, Monterey, CA, February 2005. ACM.
- [21] 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, September 2003. Springer-Verlag.
- [22] 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, April 2004. IEEE.
- [23] E. Spitznagel, D. Taylor, and J. Turner. Packet Classification using Extended TCAMs. In *IEEE International Conference on Network Protocols*. IEEE, 2003.
- [24] 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.